CHIP MULTIPROCESSORS WITH ON-CHIP AGGREGATE FUNCTION NETWORK

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Soohong P. Kim

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2009

Purdue University

West Lafayette, Indiana

dedication...

ACKNOWLEDGMENTS

To be added...

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

ABBREVIATIONS x

GLOSSARY xi

ABSTRACT xii

1 Introduction 1

1.1 Motivation 1

1.2 Our Approach 2

1.3 Related Work 4

1.3.1 Dedicated Hardware for Barriers and Collectives 4

1.3.2 Other Hardware Support for Barriers 4

1.3.3 Off-chip Aggregate Function Network for COW 5

1.3.4 Operand Networks 5

1.3.5 MIMD ISA Extensions 5

1.4 Contributions 6

1.5 Organization of this Dissertation 6

2 Parallel Communication Models 7

2.1 Shared Memory Communication Model 7

2.2 Message Passing Communication Model 11

2.3 Synchronous Aggregate Communication Model 14

3 On-Chip Synchronous Aggregate Communication Model and ISA Extensions 18

3.1 On-Chip Synchronous Aggregate Communication Model 18

3.1.1 Overview 18

3.1.2 Thread Migration Support 20

Page

3.1.3 Blocking vs. Busy-waiting AFN*OP*RD Instructions 20
3.1.4 Security 22
3.1.5 Comparison 23
3.2 ISA Extensions for the On-Chip AFN 24
3.2.1 AFN Programming Environment 24
3.2.2 AFN Instructions 24
4 On-Chip AFN Architecture 27
4.1 Baseline CMP-AFN Architecture 27
4.2 AFU-core Interconnect 27
4.3 Aggregate Function Unit 29
4.3.1 AFU Tables 30
4.3.2 AFU State Machines 30
4.3.3 Aggregate Function ALUs 31
4.3.4 AFU Allocation Table 33
4.4 A Case for CMP-AFN 33
4.4.1 Barrier Synchronization Latency for CMP-AFN 33
4.4.2 Barrier Synchronization Latency in a conventional CMP . . 35
5 Evaluation Methodology and Results 38
5.1 Simulation Target Configurations 38
5.2 Re-targeting OpenMP Benchmarks for CMP-AFN 40
5.2.1 `barrier` Construct 42
5.2.2 `critical` Construct 43
5.2.3 OpenMP Locks 43
5.3 Compiling OpenMP Benchmarks for CMP Simics Targets: Checkpointing 44
5.4 Overview of OpenMP Benchmarks 46
5.5 EPCC Microbenchmark 48
5.5.1 syncbench.c: testbar() for barrier synchronization 49

Page

5.5.2 syncbench.c: testlock() for OpenMP locks 49
5.5.3 syncbench.c: testred() for reduction 51
5.6 SPEC OMP2001 Benchmark Suite 51
5.6.1 316.applu_m 57
5.6.2 320.equake_m 57
5.6.3 324.apsi_m 57
5.6.4 330.art_m 57
5.6.5 332.ammp_m 58
5.7 Performance Evaluation 58
5.7.1 Performance Evaluation of 316.applu_m 61
6 Summary 68
6.1 Conclusions 68
6.2 Future Work 69
LIST OF REFERENCES 70
A AFN Extensions to IA-32 74
A.1 Overview of ISA Extensions for on-chip AFN 74
A.2 AFN Programming Environment 76
A.2.1 AFU 77
A.2.2 AFNreg Registers 78
A.2.3 AFNSAR Register 78
A.2.4 AFNCSR Control and Status Register 80
A.3 AFN Instructions 82
A.3.1 Opcodes 83
A.3.2 Instruction Format 83
A.3.3 Opcode Column in the Instruction Summary Table 83
A.3.4 Instruction Column in the Instruction Summary Table . . . 85
VITA 104

LIST OF TABLES

Table		Page
3.1	AFN Instructions by Group	25
4.1	Packets to communicate between CPU cores and on-chip AFN	29
4.2	A Case for CMP-AFN	34
4.3	GOMP latencies	35
4.4	barr sync latency comparison	36
5.1	Configurations for Simulated Targets	41
5.2	Replacing OpenMP constructs with AFN Library routines	42
5.3	Static Counts of OpenMP Constructs and Run-Time Routines in SPEC OMP Benchmarks for Evaluation	48
5.4	OpenMP constructs used in SPEC OMPM2001 benchmark suite: `BARR` indicates for `BARRIER` construct, `CRIT` for `CRITICAL` construct, `LOCK` for OpenMP locks, `RED` for `REDUCTION` clause, and `i.barr` for implicit barrier synchronization. The numbers are static counts.	54
5.5	SPEC OMPM2001 Benchmark Suite Description	55

LIST OF FIGURES

Figure		Page
1.1	A generic CMP-AFN architecture: The on-chip AFN consists of *AFU* (aggregate function unit) and *AFU-core Interconnect*.	3
2.1	`pi`: a sample algorithm that computes the value of Pi [30]	8
2.2	An OpenMP version of `pi`	10
2.3	An MPI version of `pi` [30]	12
2.4	Another MPI version of `pi` using collective communication [30]	13
2.5	Differences between the message-passing communication model (shown on the left) and the synchronous aggregate communication model	15
2.6	An AFAPI version of `pi` using aggregate reduction function [30]	16
2.7	A UPC version of pi.c	17
3.1	AFN Checkout Instruction with Blocking	21
3.2	AFN Checkout Instruction with Busy-Waiting	22
3.3	Differences Between Cache-Coherent Shared Memory CMP without and with the on-chip AFN	23
4.1	A Baseline CMP architecture with on-chip AFN, similar to Larrabee, where the ring network is the interprocessor network that connects multiple cores and L2 cache banks. The number of cores and L2 cache banks are implementation-dependent. (Adapted from Seiler et. al 2008)	27
4.2	On-Chip AFN Architecture: On-chip AFN consists of AFU and AFU-core Interconnect.	28
4.3	State Machine for Barrier Synchronization (AFNBARR/AFNBARRRD)	31
4.4	State Machine for Reduction Operation (AFN*OP*/AFN*OP*RD)	32
4.5	gomp_barrier_wait() when all four threads arrived a barrier at the same time	37
5.1	A four-core CMP-AFN Simics Target	39
5.2	Steps to create executables for CMP-AFN and CMP-REF targets	45

Figure Page

5.3 Structure of a Program with magic instructions 47

5.4 `delay()`: EPCC microbenchmark 49

5.5 `testbar()`: EPCC microbenchmark for barrier synchronization 50

5.6 `testlock()`: EPCC microbenchmark for OpenMP Lock. OUTERREPS=10, innerreps=128, and delaylength=500. 52

5.7 `testred()`: EPCC microbenchmark for reduction 53

5.8 ssor.f from SPEC OMP 316.applu_m 56

5.9 Summary of EPCC OpenMP Microbenchmark Results 59

5.10 Summary of SPEC OMP Benchmark Results 60

5.11 SPEC OMP 316.applu_m 61

5.12 SPEC OMP 316.applu_m (continued) 62

5.13 SPEC OMP 316.applu_m (continued) 63

5.14 applu: Breakdown of Execution Time by Cache Misses 65

5.15 applu: Instruction Count Distribution Across Cores 66

5.16 applu: Instruction Count Distribution Across Cores (Normalized to Max Instruction Count) 67

A.1 AFN Programming Environment: a single thread's perspective 75

A.2 AFNregs and AFUs 77

A.3 AFNCSR Control/Status Register 79

A.4 AFNCSR register bit positions 81

A.5 An Example Opcode Summary Table: consists of three columns – "Opcode", "Instruction", and "Description" 84

ABBREVIATIONS

AFAPI	Aggregate Function API Application Programming Interface
AFN	Aggregate Function Network
AFU	Aggregate Function Unit
CMP	Chip Multiprocessor

GLOSSARY

GOMP	An OpenMP implementation for GCC (GNU Compiler Collection). GCC version 4.2 and later.
Simics host	Refers to the computer on which one is running Simics.
Simics target	Refers to the computer simulated by Simics.

ABSTRACT

Kim, Soohong P. Ph.D., Purdue University, August, 2009. Chip Multiprocessors with On-Chip Aggregate Function Network. Major Professors: Samuel P. Midkiff and Henry G. Dietz.

State-of-the-art on-chip networks and block-based cache coherence protocols used in cache-coherent shared-memory Chip MultiProcessors (CMPs) are inefficient for collective operations across cores. Performance of CMPs can be seriously degraded by the multitude of memory requests and coherence messages required to implement each collective operation. This thesis presents a CMP-AFN architecture and Instruction Set Architecture (ISA) extensions that augment a conventional shared-memory CMP with a tightly-integrated Aggregate Function Network (AFN) that implements low-latency collective operations without using or interfering with the memory hierarchy. For a modest increase in circuit complexity, traffic within a CMP's internal network is dramatically reduced, improving the performance of caches and reducing power consumption. Full system simulations of 16-core CMPs show a CMP-AFN outperforms the reference design significantly, eliminating up to 52% of memory accesses and up to 73% of private L1 data cache misses in both the EPCC OpenMP microbenchmarks and SPEC OMP benchmarks.

1. INTRODUCTION

1.1 Motivation

Chip Multiprocessors (CMPs) dominate the important desktop, server and HPF market segments because they allow faster application performance using shared memory programming models. Application studies show, however, that synchronization overhead is the performance bottleneck in parallel applications for cache-coherent shared memory multiprocessors [6–8]. Synchronization operations often account for significant fractions of execution time and can both limit the scalability of parallel programs on very large machines and negatively affect the ability of large-scale systems to exploit fine-grain parallelism. Sampson, et al. [9] shows the importance of fast barrier synchronization in shared-memory many-core CMPs as a prerequisite for the exploitation of fine-grained parallelism.

Despite enabling lower latency for interprocessor communication, current state-of-the-art on-chip networks and block-based cache coherence protocols for shared memory multi-core architectures are inefficient for collective communication [10]. Multiple memory requests and coherence messages must be transmitted among CPU cores and cache controllers to implement a single collective operation. Software synchronization primitives include atomic read-modify-write (RMW) instructions, which are on the critical path of the synchronization algorithm. When these instructions are implemented with shared variables, they interfere with the cache coherence protocol and generate a significant amount of network traffic because of contention for synchronization flags. Coherency-related overheads are not the only ones suffered. Aslot, et al. [6] shows that there are two aspects of lock overhead: the overhead of executing extra instructions while spin waiting and the overhead of acquiring the lock. For

scalable CMP performance, the negative impact of both aspects must be reduced or eliminated.

1.2 Our Approach

The solution we propose is the adoption and adaptation of the synchronous aggregate communication model [11], which was initially developed for the cluster of workstations (COW). The synchronous aggregate communication model can be implemented within a CMP with the on-chip Aggregate Function Network (AFN) and ISA extensions. In this dissertation, we detail the new CMP-AFN architecture and the corresponding instruction set architecture (ISA) extensions that augment a shared memory chip multiprocessor with an aggregate function network and an interface between CPU cores and the on-chip AFN.

In the CMP-AFN architecture (shown in Figure 1.1), collective communication is performed without using or interfering with the on-chip cache coherent shared memory hierarchy. Collective communication can be performed via on-chip AFN, a dedicated network or an embedded virtual network in an existing on-chip interconnect. Because spin-wait on the shared variables for the synchronization primitives can be avoided in the CMP-AFN, the excessive coherence messages are eliminated in the on-chip interconnect and CPU time waste can be avoided. As a result, CMP-AFN provides low latency collective operations and reduces coherence traffic, resulting in effective use of on-chip cache and low power consumption.

The goal of our research is to explore the synchronous aggregate communication model in cache-coherent shared-memory chip multiprocessors to exploit multithreaded applications. We focus primarily on the parallel execution model associated with OpenMP, but our approach is applicable to any shared-memory programming model that supports barriers and collective operations such as reduction.

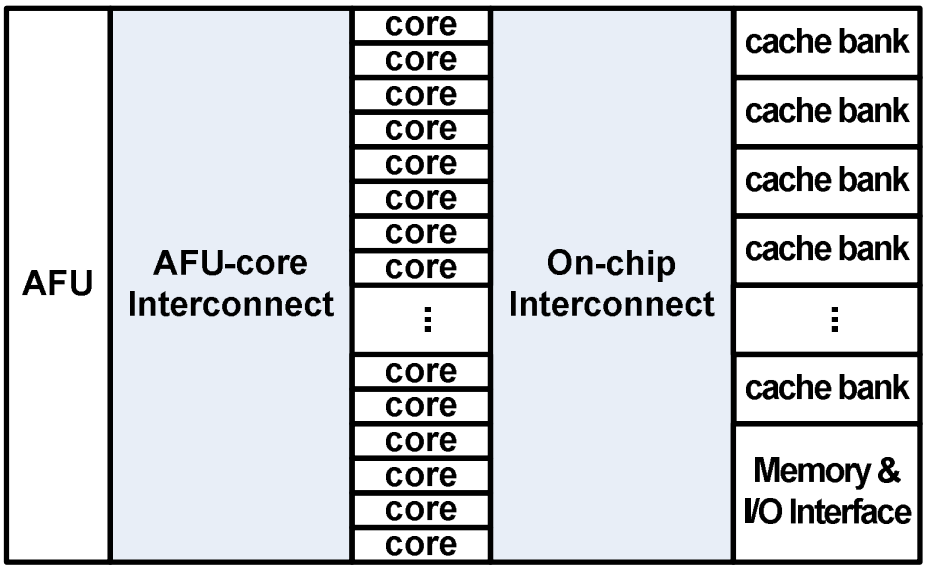


Fig. 1.1. A generic CMP-AFN architecture: The on-chip AFN consists of *AFU* (aggregate function unit) and *AFU-core Interconnect*.

1.3 Related Work

We now discuss prior work related to this dissertation, organized by topic.

1.3.1 Dedicated Hardware for Barriers and Collectives

The NYU Ultracomputer [12] and the IBM RP3 [13] are shared memory architectures in which the multistage interconnection networks that combine multiple messages that reference identical memory location. The Cray T3D [14] directly supports barrier synchronization, swap, and Fetch-and-Increment. The CM-5 Connection Machine [15] has a control network that supports reduction operations, prefix operations, maximum, logical OR, and XOR. Cedar [16] and Alliant FX/8 supported synchronization hardware. Tera and Denelcor HEP also supported synchronization hardware. The IBM BlueGene/L [17] has three types of networks that make up the interprocessor "fabric". In addition to a Torus interconnect for the point-to-point messages, it has a collective network for one-to-all broadcast and collective operations, and a barrier network for barrier synchronization. The proposed CMP-AFN architecture features the dedicated on-chip hardware for barrier synchronization and collective communication within a single chip multiprocessor.

1.3.2 Other Hardware Support for Barriers

A barrier filter [9] is a hardware mechanism for the CMP architecture for barrier synchronization that does not rely on locks nor busy waiting. Instead, it starves processing elements' requests to cache lines until all processing elements arrive at the barrier, then fills a cache line at the synchronization point. A barrier filter does not provide an effective mechanism for collective communication. Although it does not require ISA extensions or CPU core modification, address tag filtering might be in the critical path for all memory accesses, not just barrier synchronization, and hence can potentially increase the latency for all memory accesses.

1.3.3 Off-chip Aggregate Function Network for COW

PAPERS [18] is a custom network hub that is attached to a cluster of workstations (COW). This *off-chip* aggregate function network design uses a combination of barrier synchronization with a four-bit wide global NAND to construct a robust library of aggregate functions. While PAPERS is off-chip aggregate function networks for the cluster architecture, the on-chip AFN presented in this thesis is for the shared memory multi-core architectures and features the architectural support for thread migration by operating systems.

1.3.4 Operand Networks

A variety of architectures have been proposed to support and utilize *operand networks*, including RAW [19], TRIPS [20], and WaveScalar [21]. Operand networks communicate register values between consumer and producer instructions. The on-chip AFN can be considered as an operand network to communicate register values among multiple producer instructions and multiple consumer instructions for synchronous collective communication, that are performed on the network.

1.3.5 MIMD ISA Extensions

There has been much research on ISA extensions to support various forms of hardware multithreading. Multiple Instruction Stream Computers (MISC) [22] is a message-passing based hardware mechanism where the parallel execution of multiple instruction streams can be orchestrated. MISC supports point-to-point communication between processing elements and between PEs and the memory subsystem. Multiple Instruction Stream Processing (MISP) architecture [23] provides inter-sequencer signaling and an asynchronous control transfer mechanism.

1.4 Contributions

We make the following major contributions:

- We introduce the synchronous aggregate communication model to chip multiprocessor architectures to provide low-latency collective operations and reduced on-chip network traffic, resulting in the effective use of on-chip cache and low power consumption.
- We describe on-chip aggregate function network architecture and associated ISA extensions that implement the synchronous aggregate communication model and allow the operating system to schedule threads to one CPU core to another during the execution of a collective operation.
- We provide experimental results using full-system simulation of CMPs showing the performance benefits of the proposed techniques are large. In an 8-core CMP, we see speedups of up to 35% on the SPEC OMP benchmarks, with benefits increasing with higher core count.

1.5 Organization of this Dissertation

Chapter 2 discusses various parallel communication models. Chapter 3 presents the on-chip synchronous aggregate communication model and ISA (Instruction Set Architecture) extensions for the aggregate communication model. Chapter 4 describes the on-chip AFN architecture and CMP-AFN architecture. Chapter 5 presents the evaluation methodology and experiment results. Finally, Chapter 6 provides conclusions and future work.

2. PARALLEL COMMUNICATION MODELS

In this chapter, we discuss various parallel communication models for multi-core architecture, i.e. shared memory communication model, point-to-point message passing communication model, and synchronous aggregate communication model.

Although we do not discuss in details, but there are more intra-socket parallel communication models for the architectures such as 80-core Intel Terascale processors [24] (point-to-point communication), IBM Cell BE-based systems [25] [26] (point-to-point combined with DMA), and PGAS (Partitioned Global Address Space) on Multi-Cores [27] [28] ("one-sided" communication, i.e. load/store on shared memory communication model).

In addition to comparison of these parallel communication model, we present a representative programming language (OpenMP, MPI, AFAPI, and UPC) and a sample program for each parallel communication discussed here. We selected a sample algorithm to compare various approaches. Shown in Figure 2.1, the algorithm computes the value of Pi and has been used in various publications [29] [30] to demonstrate a variety of different parallel programming environments.

2.1 Shared Memory Communication Model

Shared Memory is a model for interactions between processing elements (PEs) within a parallel system. Shared memory systems include CMP (Chip Multi-Processors), SMP (Symmetric Multi-Processors), or DSM (Distributed Shared Memory). PEs in shared memory systems share a (either logically or physically) single memory and a value written to shared memory by one PE can be directly accessed by other PEs. Shared memory is generally considered easier to program than message passing, but

```
#include <stdlib.h>
#include <stdio.h>
main(int argc, char **argv) {
  register double width, sum;
  register int intervals, i;

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  /* do the computation */
  sum = 0;
  for (i=0; i<intervals; ++i) {
    register double x = (i + 0.5) * width;
    sum += 4.0 / (1.0 + x * x);
  }
  sum *= width;
  printf("Estimation of pi is %f\n", sum);
  return(0);
}
```

Fig. 2.1. `pi`: a sample algorithm that computes the value of Pi [30]

it requires on-chip cache-coherence protocol, which requires very high development cost and time-to-market.

OpenMP

OpenMP [31] [32] [33] is an industry standard set of pragmas, environment variables, and a run-time library that tell the compilers (C++ and Fortran) when, where, and how to create multithreaded code for shared memory multiprocessors. With OpenMP, programmers tell the compiler what to do with threads at an abstract level and leave the low-level details (such as thread management) to the compiler. This approach makes OpenMP much easier to use than Pthreads, but at the expense of some control and some performance. Unlike MPI, no explicit communication is used, instead inter-thread communication is done implicitly via shared memory with load-/store instructions. As it is limited to the shared memory architectures, OpenMP for non-shared memory multiprocessors have been proposed [34] [35]. Figure 2.2 shows an OpenMP version of the `pi.c` algorithm.

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
  register double width, sum, x;
  register int i, intervals;

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / (double) intervals;

  /* do the local computations, followed by reduction */
  sum = 0;
#pragma omp parallel for reduction(+:sum) private(x)
  for (i=0;i< intervals; i++){
    x = (i + 0.5) * width;
    sum += 4.0 / (1.0 + x * x);
  }

  /* have only the master thread print the result */
#pragma omp master
  {
    /* scale by width */
    sum = sum * width;
    printf("Estimation of pi is %14.12lf\n", sum);
  }
  exit(0);
}
```

Fig. 2.2. An OpenMP version of `pi`

2.2 Message Passing Communication Model

Message Passing is a model for interactions between processing elements (PEs) within a parallel system, typically cluster architectures. In general, a message is constructed by software on one PE and is sent through an interconnection network to another PE, which then must accept and act upon the message contents. Although the overhead in handling each message (latency) may be high, there are typically few restrictions on how much information each message may contain. Thus, message passing can yield high bandwidth making it a very effective way to transmit a large block of data from one PE to another. In the same token, this point-to-point communication model is not efficient for collective communication because it requires global communications from all PEs with relatively smaller data. On the other hand, point-to-point communication does not require the expensive hardware support for cache coherent protocol. Therefore, it is easier and faster to build the multi-core architectures that only requires point-to-point communication model.

MPI

The Message Passing Interface (MPI) [36] [37] is a language-independent communications API (Application Programming Interface) for message passing on cluster architectures. It expresses parallelism explicitly rather than implicitly. MPI is successful in achieving high scalability and high portability thanks to the underlying hardware architecture. However, MPI requires more programming efforts compared to OpenMP as programmers need to explicitly schedule inter-PE communication.

Figure2.3 shows an MPI program for the pi algorithm that uses basic MPI message-passing calls for each PE to send its partial sum to PE 0, which sums and prints the result. Figure 2.4 shows another MPI version for the pi algorithm that uses collective communication (which, for this particular application, is clearly the most appropriate).

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv) {
  register double width;
  double sum, lsum;
  register int intervals, i;
  int nproc, iproc;
  MPI_Status status;

  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;
  lsum = 0;
  for (i=iproc; i<intervals; i+=nproc) {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
  lsum *= width;
  if (iproc != 0) {
    MPI_Send(&lbuf, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  } else {
    sum = lsum;
    for (i=1; i<nproc; ++i) {
      MPI_Recv(&lbuf, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
               MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      sum += lsum;
    }
    printf("Estimation of pi is %f\n", sum);
  }
  MPI_Finalize();
  return(0);
}
```

Fig. 2.3. An MPI version of `pi` [30]

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv) {
  register double width;
  double sum, lsum;
  register int intervals, i;
  int nproc, iproc;

  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;
  lsum = 0;
  for (i=iproc; i<intervals; i+=nproc) {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
  lsum *= width;
  MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
             MPI_SUM, 0, MPI_COMM_WORLD);
  if (iproc == 0) {
    printf("Estimation of pi is %f\n", sum);
  }
  MPI_Finalize();
  return(0);
}
```

Fig. 2.4. Another MPI version of `pi` using collective communication [30]

2.3 Synchronous Aggregate Communication Model

Dietz, et al. [11, 38] proposed the *synchronized aggregate communication model* for clusters of workstations, implemented using off-chip custom network hardware and the associated API. The communication model is neither message-passing nor shared memory, but is based on the concept of synchronously aggregating data from a group of processing elements. As shown in the right side of Figure 2.5, the off-chip aggregate function network collects an aggregate of the data items from all processing elements in the current barrier, performs a synchronous collective operation on the aggregated data, and then each processing element obtains the result of the operation from the aggregate function network. The network hardware, an *aggregate function network*, is connected to the workstations via parallel ports and provides a cluster of workstations with a fast barrier synchronization mechanism. Immediately after the barrier synchronization, processing elements may additionally perform a communication operation, called a *synchronous aggregate communication* or a *synchronous collective operation*. This communication model is efficient for collective operations because the communication is not point-to-point and the computation is done on the network. In the message passing version of the operation, shown in the left side of Figure 2.5, data must be communicated to processors in several steps, with accumulated totals being formed in each step.

Collective communication is another name for aggregate functions, most often used when referring to aggregate functions that are constructed using multiple message-passing operations.

AFAPI

The Aggregate Function API (AFAPI) library [38] was initially designed for the aggregate function clusters, such as various types of PAPERS clusters. Later it was ported to shared memory multiprocessors and clusters of Linux systems. AFAPI provides parallel systems with a model that offers a good target for compilers by

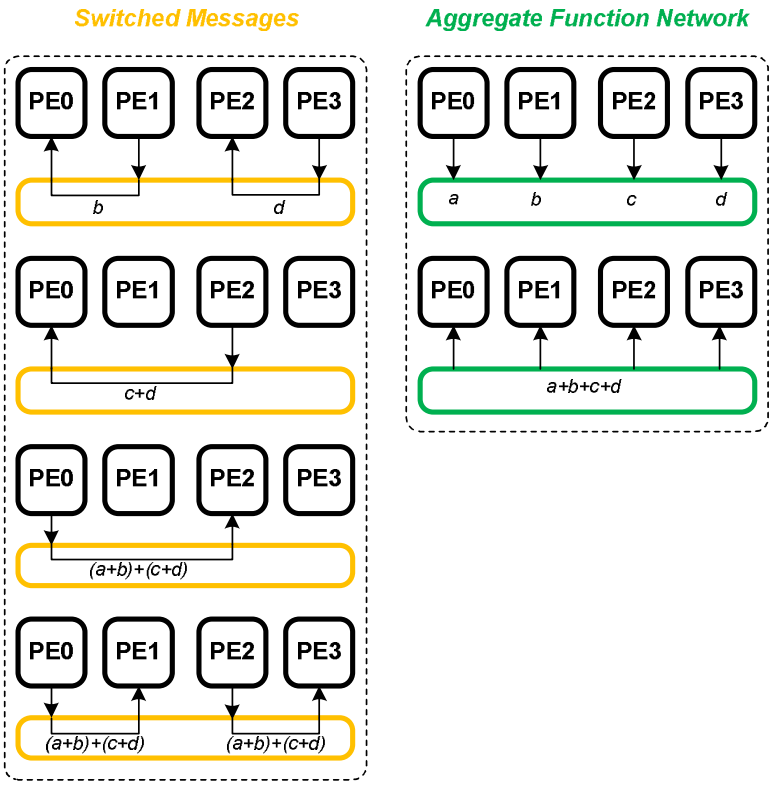


Fig. 2.5. Differences between the message-passing communication model (shown on the left) and the synchronous aggregate communication model

combining low latency with predictable performance. Figure 2.6 shows an AFAPI version of the `pi`.

```
#include <stdlib.h>
#include <stdio.h>
#include <AFAPI/afapi.h>

int main(int argc, char **argv) {
    register double width, sum;
    register int intervals, i;

    /* check-in with AFAPI */
    if (p_init()) exit(1);

    /* get the number of intervals */
    intervals = atoi(argv[1]);
    width = 1.0 / intervals;

    /* do the local computations */
    sum = 0;
    for (i=IPROC; i<intervals; i+=NPROC) {
        register double x = (i + 0.5) * width;
        sum += 4.0 / (1.0 + x * x);
    }

    /* sum across the local results & scale by width */
    sum = p_reduceAdd64f(sum) * width;

    /* have only the console PE print the result */
    if (IPROC == CPROC) {
        printf("Estimation of pi is %14.12lf\n", sum);
    }

    /* check-out */
    p_exit();
    exit(0);
}
```

Fig. 2.6. An AFAPI version of `pi` using aggregate reduction function [30]

```
#include <stdlib.h>
#include <stdio.h>
#include <upc_relaxed.h>

upc_lock_t *lock0;
shared double pi;

int main(int argc, char **argv) {
  register double width, sum;
  register int intervals, i;

  lock0 = upc_all_lock_alloc();

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  upc_barrier;
  /* do the local computations */
  sum = 0;
  upc_forall (i=0; i<intervals; i++; i) {
    register double x = (i + 0.5) * width;
    sum += 4.0 / (1.0 + x * x);
  }

  /* sum across the local results & scale by width */
  pi = 0;
  upc_lock(lock0);
  pi += sum * width;
  upc_unlock(lock0);

  /* ensure all is done */
  /* have only the console PE print the result */
  upc_barrier();
  if(MYTHREAD==0)
    printf("Estimation of pi is %14.12lf\n", pi);

  upc_lock_free(lock0);
  exit(0);
}
```

Fig. 2.7. A UPC version of pi.c

3. ON-CHIP SYNCHRONOUS AGGREGATE COMMUNICATION MODEL AND ISA EXTENSIONS

In this chapter, we describe the proposed on-chip synchronous aggregate communication model and instruction set architecture (ISA) extensions for on-chip aggregate communication model.

3.1 On-Chip Synchronous Aggregate Communication Model

The on-chip synchronous aggregate communication model is intended for a single socket processor with multiple cores, where a team of multiple threads are controlled by an operating system.

The major difference between aggregate communication models for off-chip and on-chip is the thread migration support. The on-chip synchronous aggregate communication model allows a team of threads to access the on-chip aggregate function network from any cores in the system.

We now provide an overview of the on-chip aggregate communication model in the context of the SPMD (Single Program Multiple Data) parallel execution model, where a team of threads executes a single program.

3.1.1 Overview

A thread using an aggregate function network performs a synchronous collective operation in three phases: (1) an initiation phase, to request a synchronous collective operation and send data to that operation, (2) a waiting phase, during which a thread waits for the result of the operation to be available and then receives that operation,

and (3) a completion phase, to acknowledge the receipt of the result and to inform the AFN that the thread is about to proceed past the operation.

A team of threads requests, and is assigned, a unique team number (afuID) by the AFN. Prior to any collective operation, each thread is also assigned a unique thread number (rank) within a team. For synchronous collective communication operations, each thread executes a pair of instructions: *checkin* and *checkout*. The checkin instruction is of the form AFN*OP*, where *OP* specifies a collective operation such as a barrier synchronization or a reduction. The checkout instruction is of the form AFN*OP*RD, where *OP* is as before. Use of checkin and checkout instructions for a single collective operation allows other instructions to be scheduled between the checkin and checkout instructions (i.e., the initiation and completion phases), and, as explained shortly, allows a participating thread to execute checkin and checkout instructions in different cores, enabling thread migration.

When a thread executes an AFN*OP* checkin instruction, the CPU core sends the opcode and any needed data to the AFN to initiate the synchronous collective operation. When a thread executes the AFN*OP*RD checkout instruction, the CPU core sends its *core id* (cores ids are from $0 \ldots number\ of\ cores - 1$) to the AFN, and then waits for the result to be sent by the AFN. Upon the receiving the result from the AFN (as the result of a checkout instruction), the CPU core sends an acknowledgment to the AFN, and the synchronous operation is finished.

Each team of threads is assigned a `secureID` by the operating system that is used to ensure that a thread initiating or participating in an AFN operation on some AFU is part of the team that is associated with the AFU. **s***ecureID*, **a***fuID*, and the **r***ank* constitutes a so-called *SAR address*, which serves as an implicit operand for the AFN*OP* and AFN*OP*RD instructions. The SAR address is sent to the on-chip AFN along with opcode and other explicit operands of AFN instructions. Both the *SAR* and use of split functions allow the AFN to be safely used in a multi-programmed, preemptively scheduled, system.

3.1.2 Thread Migration Support

In a multicore system, a thread may be initially assigned to one CPU core and later be migrated to another CPU core by the OS. This migration may occur during the execution of a collective communication. To support thread migration, a pair of AFN instructions (checkin and checkout instructions) is used for each collective communication. First, a thread initiates a collective communication with an AFN*OP* checkin instruction, and then obtains the result of the collective communication from the AFN with an AFN*OP*RD checkout instruction. The checkin and checkout instructions do not have to be executed in the same core. The checkout instruction allows the AFN to send the result of the collective communication to the core hosting the thread that executes the checkout instruction.

The on-chip AFN does not need to track the migration of threads across cores, nor does it need to "wake-up" sleeping threads. When the operating system context-switches out a stalled thread that is waiting for the result of an AFN operation while executing the AFN*OP*RD instruction, the AFN*OP*RD instruction will not have committed. After the thread is scheduled (perhaps on another core), it will need to re-issue the AFN*OP*RD instruction. When a thread receives the result from the AFN, the AFN*OP*RD instruction will be retired.

Whether the checkout instruction is implemented as blocking or busy-waiting, it may be possible that the result from the on-chip AFN may arrive at the core after a thread has been moved from the core by the OS. In order to provide the migrated thread with the result when it is rescheduled, the on-chip AFN is not allowed to complete the current operation and the CPU core is required to send the acknowledgment to the on-chip AFN upon the receipt of the result.

3.1.3 Blocking vs. Busy-waiting AFN*OP*RD Instructions

Types of synchronization can be either blocking or busy-waiting. Busy-waiting is preferable when the scheduling overhead is larger than expected wait time, or proces-

```
afn_barrierSync_blocking() {
    asm volatile("afnbarr_cin");
    asm volatile("afnbarr_cout_block");
}
```

Fig. 3.1. AFN Checkout Instruction with Blocking

```
afn_barrierSync_busywaiting() {
    asm volatile("afnbarr_cin");
    while (cond)
       asm volatile("afnbarr_cout_busywait");
}
```

Fig. 3.2. AFN Checkout Instruction with Busy-Waiting

sor resources are not needed for other tasks. For the on-chip aggregate communication model, the checkout instructions can be implemented using either blocking or busy-waiting, as shown in Figure 3.1 and Figure 3.2, respectively. Busy-waiting might be more flexible and may allow for intelligent backoff. Unlike cache coherent shared memory, busy-waiting actually increases the network traffic between the on-chip AFN and the core. On the other hand, blocking will significantly reduce the network traffic and dynamic instruction count. For our experimental results we have assumed the busy-waiting implementation. For the performance evaluation, we use busy-waiting due to easier implementation in the simulation environment.

3.1.4 Security

A team of threads is assigned a `secureID` by the OS using privileged instructions. The `secureID` is stored in the `AFNSAR` register. The contents of `AFNSAR` register can only be manipulated and read by privileged instructions. When a thread initiates an AFN operation, requests the result, or sends an acknowledgment to the on-chip AFN, `secureID` is sent to the on-chip AFN for authentication and must match the value at the on-chip AFN. Should the `secureID` not match, the AFU will throw an exception. The `secureID` prevents a buggy or malicious thread from another process participating in a communication operation that it should not be a part of.

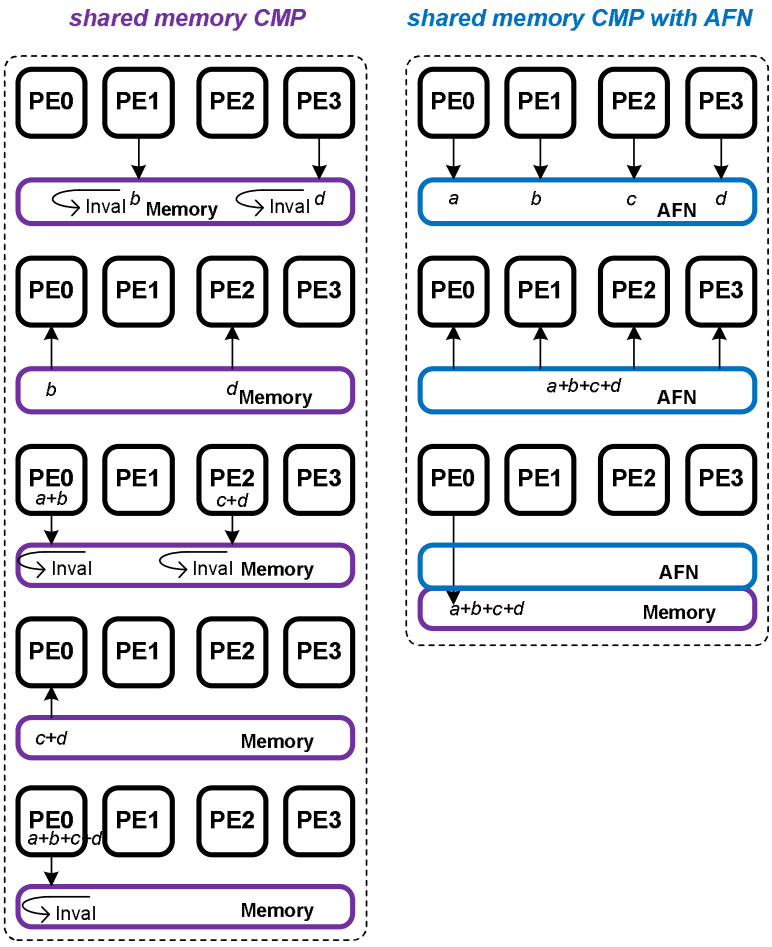


Fig. 3.3. Differences Between Cache-Coherent Shared Memory CMP without and with the on-chip AFN

3.1.5 Comparison

Figure 3.3 shows differences in the execution of collective operation (reduction) between the on-chip shared memory communication model and the on-chip synchronous aggregate communication model. With the on-chip aggregate communication model, the collective operation does not generate coherence protocol messages and does not require PE's involvement for the computation.

3.2 ISA Extensions for the On-Chip AFN

In this section, we describe ISA extensions for the on-chip AFN (*AFN extensions* for short) to the Intel IA32 architecture [39]. The AFN extensions do not require a specific base ISA, but should be applicable to an arbitrary ISA. AFN extensions use the synchronous aggregate communication model [11]. The extensions support inter-processor data communication, barrier synchronization, and synchronous collective operations via an on-chip aggregate function network. The details of the ISA extensions are described in the Appendix A.

3.2.1 AFN Programming Environment

All AFN instructions use general-purpose registers or floating-point registers for both source and destination registers. The AFN extensions provide the following resources: eight 32-bit General Purpose Registers, eight 80-bit Floating Point Data Registers, the 32-bit AFNSAR Address Register (added for the AFN extensions) and a 32-bit AFNCSR Control/Status Register (added for the AFN extensions), which contains status and control bits used in such as AFN floating-point operations.

3.2.2 AFN Instructions

As shown in Table 3.1, AFN instructions are divided into five functional groups – configuration, synchronization, reduction, data movement, and state management.

AFN Configuration Instructions include `AFUALLOC`, which allocates an AFU for a team of threads, and `AFUFREE`, which frees up the AFU. These instructions can only be executed in privileged mode and associate and disassociate thread teams from an AFU.. The first operand of `AFUALLOC` represents the number of threads for the team. When a new team of threads is created, an AFU is assigned using the `AFUALLOC` instruction. The on-chip AFN returns the `afuID` to the CPU core that executes the `AFUALLOC` instruction. When no AFU is available for the new team, the on-chip AFN

Table 3.1
AFN Instructions by Group

AFN Configuration Instructions	
`AFUALLOC`	Allocate an AFU
`AFUFREE`	Free up an AFU
Synchronization Instructions	
`AFNBARR`	Barrier Synchronization
`AFNBARRRD`	Barrier Synchronization Read
`AFNLOCK`	Mutex Lock
`AFNUNLOCK`	Mutex Unlock
Reduction Instructions	
`AFNADD/AFNFADD/AFNFADDP`	Reduce ADD
`AFNMUL/AFNFMUL/AFNFMULP`	Reduce MULTIPLY
`AFNAND`	Reduce Bitwise AND
`AFNOR`	Reduce Bitwise OR
`AFN`*OP*`RD`	Read Result
Data Movement Instructions	
`AFNPUT/AFNGET`	Multi-broadcast Data Communication
State Management Instructions	
`LDAFNCSR`	Load AFNCSR Register
`STAFNCSR`	Store AFNCSR Register State
`LDAFNSAR`	Load AFNSAR Register
`STAFNSAR`	Store AFNSAR Register State

clears the most significant bit of the `afuID`. The user AFN operations will trap if the most significant bit of the afuID is zero, then AFN operations will be performed via software emulation, not via on-chip AFN hardware. This allows virtualization of the on-chip AFU and correct execution of programs when the number of thread teams exceeds the number of AFUs in the system.

`AFUFREE` takes no operands. The CPU that executes `AFUFREE` instruction extracts the `afuID` from `AFNSAR` and then sends the `afuID` to the on-chip AFN.

The synchronization instructions are `AFNBARR` and `AFNBARRRD`, which perform a barrier synchronization and `AFNLOCK` and `AFNUNLOCK`, which perform a mutex lock and a mutex unlock using the on-chip AFN.

The reduction instructions are AFN*OP* and AFN*OP*RD, which perform a reduction operation. *OP* can be *ADD*, *MUL*, *AND*, and *OR* for reduce add, reduce multiply, bitwise AND, and bitwise OR, respectively. `AFN`*OP* copies the operand (source operand) to the on-chip AFN. For `AFNADD`, the on-chip AFN adds up the aggregated data from all threads.

Data movement instructions are `AFNPUT` and `AFNGET`, which perform a multi-broadcast `PutGet` operation. `AFNPUT` takes two operands. The first operand is the datum for the multi-broadcast and the second operand represents the rank of the thread where the data comes from.

Finally, the state management instructions are `LDAFNCSR` and `STAFNCSR`. `LDAFNCSR` loads the source operand into the `AFNCSR` control and status register. The source operand is a 32-bit memory location. The `STAFNCSR` instruction stores the contents of the `AFNCSR` register into memory.

4. ON-CHIP AFN ARCHITECTURE

4.1 Baseline CMP-AFN Architecture

Our baseline CMP architecture uses multiple CPU cores, shared L2 cache, and the ring interconnect, similar to the Larrabee many-core architecture [40], as shown in Figure 4.1. The number of cores and L2 cache banks are implementation-dependent, as is the position of the on-chip AFN.

4.2 AFU-core Interconnect

The on-chip AFN consists of the *AFU-core interconnect* and the *aggregate function units* (AFU) as shown in Figure 4.2. The AFU-core interconnect could be a dedicated interconnect or could be embedded in the existing interconnect network.

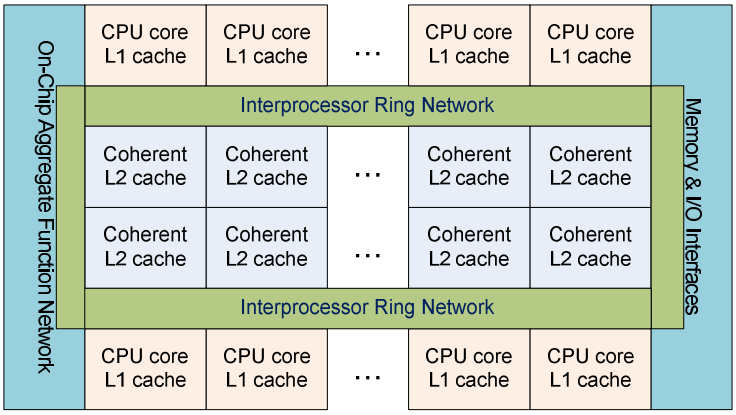


Fig. 4.1. A Baseline CMP architecture with on-chip AFN, similar to Larrabee, where the ring network is the interprocessor network that connects multiple cores and L2 cache banks. The number of cores and L2 cache banks are implementation-dependent. (Adapted from Seiler et. al 2008)

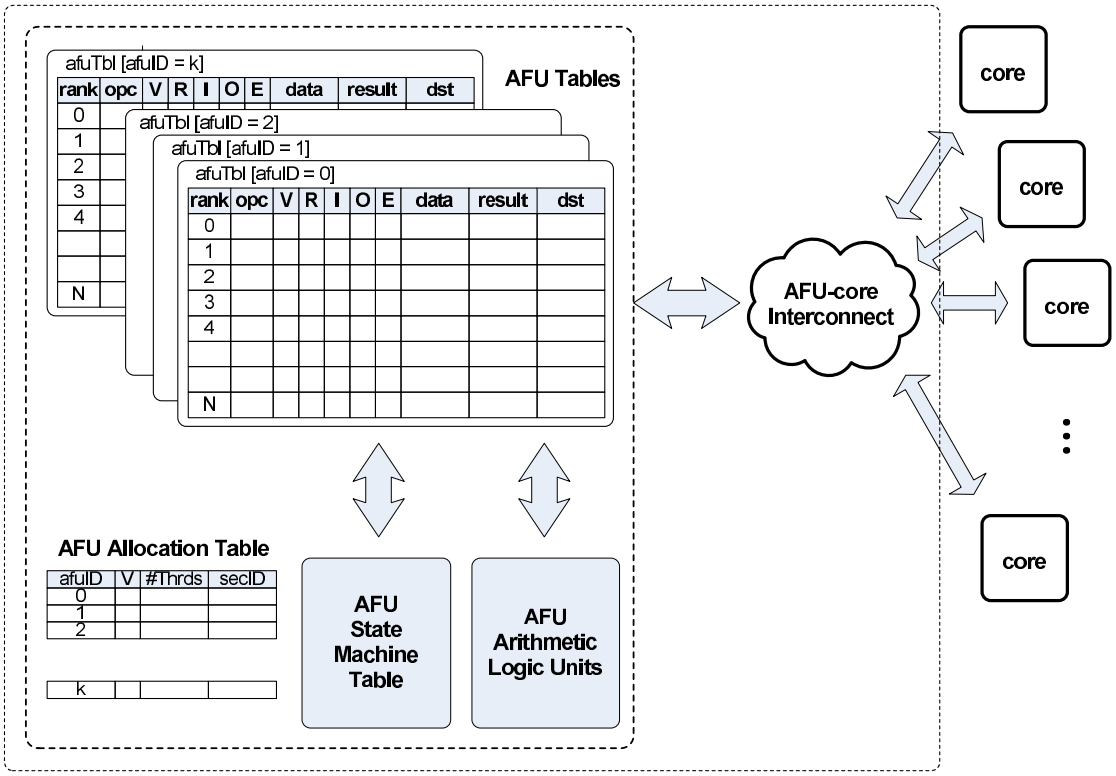


Fig. 4.2. On-Chip AFN Architecture: On-chip AFN consists of AFU and AFU-core Interconnect.

In our baseline architecture, the on-chip AFN is attached to the ring interprocessor network. The communication between the on-chip AFN and cores is done through the AFU-core interconnect. A dedicated interconnect for the AFU-core interconnect requires additional wires and logics. Alternatively, the AFU-core interconnect could be a virtual channel with the highest priority over the existing interconnect network, similar to the Cray T3E's logical barrier/eureka network [41]. The dedicated interconnect and virtual channel implementations have design trade-offs of latency versus on-chip real-estate.

The on-chip AFN and CPU cores communicate with each other via AFN request and response packets along the AFU-core interconnect. CPU cores send *AFN request packets* to the on-chip AFN. In response to the request packets, the on-chip AFN sends *AFN response packets* to the requesting CPU. As shown in Table 4.1, an AFN

Table 4.1
Packets to communicate between CPU cores and on-chip AFN

instructions	core-to-AFU packets			AFU-to-core packets	
	opcode	operand	SAR	status	result
AFUALLOC	opcALLOC	numThrds	secureID	error, complete	afuID
AFUFREE	opcFREE	-	secureID, afuID	error, complete	-
AFNBARR	opcBARR	-	secureID, afuID, rank	error, complete	-
AFNBARRRD	opcBARRRD	-	secureID, afuID, rank	error, complete, retry	-
AFNLOCK	opcLOCK	-	secureID, afuID, rank	error, complete, retry	-
AFNUNLOCK	opcUNLOCK	-	secureID, afuID, rank	error, complete, retry	-
AFNADD	opcADD	data	secureID, afuID, rank	error, complete	-
AFNMUL	opcMUL	data	secureID, afuID, rank	error, complete	-
AFNAND	opcAND	data	secureID, afuID, rank	error, complete	-
AFNRD	opcRD	-	secureID, afuID, rank	error, complete, retry	data
AFNFADD	opcFADD	data	secureID, afuID, rank	error, complete	-
AFNFMUL	opcFMUL	data	secureID, afuID, rank	error, complete	-
AFNFRD	opcFRD	-	secureID, afuID, rank	error, complete, retry	data
AFNPUT	opcPUT	data, srcThrd	secureID, afuID, rank	error, complete,	-
AFNGET	opcGET	-	secureID, afuID, rank	error, complete, retry	data
-	opcRDACK	-	secureID, afuID, rank	-	-

request packet contains the SAR address, opcode, and data. An AFN response packet contains the SAR address, status, and result. The on-chip AFN extracts appropriate information from incoming request packets, processes the packets, and creates the response packets based on the table shown in Table 4.1.

As the on-chip AFN may broadcast the result of collective communications to all cores, it may leverage the hardware support for multicast such as VCTM (Virtual Circuit Tree Multicasting) [42]. VCTM supports multicast of network messages, such as invalidation messages from the coherence protocol, in the network on chip (NoC).

4.3 Aggregate Function Unit

In this section, we describe AFU, which consists of AFU tables, state machines, aggregate ALUs and an AFU allocation table, as shown in Figure 4.2.

4.3.1 AFU Tables

Each AFU table keeps various information for a team of threads and is indexed by *afuID*, i.e. `afuTbl[afuID]`. Per-thread information in each table is indexed by thread's *rank*, i.e. `afuTbl[afuID][rank]`, and stores opcode (opc), 5-bit state (V/R/I/O/E), data, and core number. V stands for valid, R for `RESET`, I for `INPUT_ARRIVED`, O for `OUTPUT_READY`, E for `ERROR`. `afuTbl[afuID].INPUT_ARRIVED[N:0]` is an *arrived vector*, where each bit position indicates whether the corresponding thread has arrived or not.

4.3.2 AFU State Machines

AFUs use state machines to process AFN operations. An AFU maintains a state machine for each AFN operation. And for each AFN operation, one state machine exists per thread, and are updated when qualified events occur, including the arrival of an AFN opcode from the core. Figure 4.3 shows the state machine for AFNBARR/AFNBARRRD (barrier synchronization). The state machine consists of four states, `RESET`, `INPUT_ARRIVED`, `OUTPUT_READY`, and `ERROR`. `RESET` is the initial state and indicates that the corresponding thread is waiting for the request packet with opcode `opcBARR` to arrive at the AFU. Upon the arrival of `opcBARR` packet for the thread, the AFU changes the state from `RESET` to `INPUT_ARRIVED`. The `INPUT_ARRIVED` state indicates that the thread is waiting for `opcBARR` packets of other threads in a team to arrive at the AFU. When all threads arrives at the AFU (i.e., `VALID[N:0]` == `INPUT_ARRIVED[N:0]`), the states for *all* threads become `OUTPUT_READY`. The `OUTPUT_READY` state indicates that the thread is waiting for the packet with opcode `opcBARRRD` to complete the barrier synchronization. Upon the arrival of `opcBARRRD` packet, the AFU changes the state to `RESET` for the corresponding thread. `RESET` state indicates that the thread is ready to take next barrier synchronization or any AFN operations.

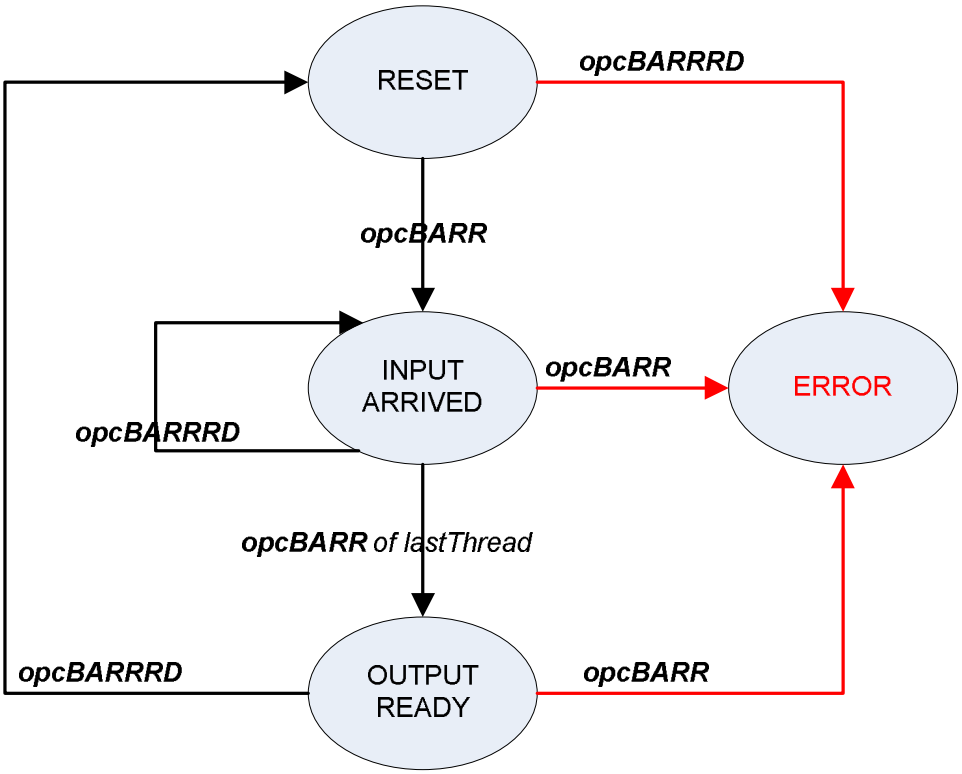


Fig. 4.3. State Machine for Barrier Synchronization (AFNBARR/AFNBARRRD)

We note that threads in a team may not all change their states back to `RESET` at the same time. However, the AFU does not prevent threads in the `RESET` state from entering the next barrier synchronization even if there are threads whose states are `OUTPUT_READY` for the previous barrier synchronization. The state machine for AFN*OP*/AFN*OP*RD is shown in Figure 4.4, which is similar to that of barrier synchronization.

4.3.3 Aggregate Function ALUs

A collective operation is an N-operand operation and can be implemented using a binary ALU with multiple iterations or using N-ary ALU with a single iteration. There is a design trade-offs. In principle, the AFU waits for all operands to arrive

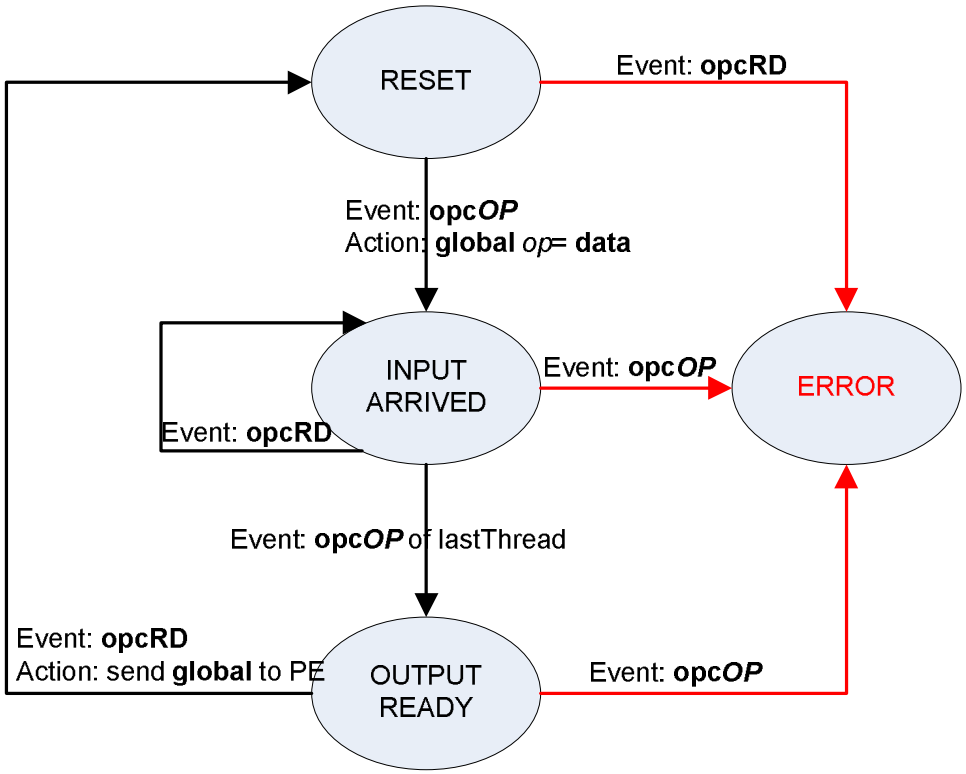


Fig. 4.4. State Machine for Reduction Operation (AFN*OP*/AFN*OP*RD)

(i.e. all threads reach synchronization point) to perform N-operand computation of the aggregate function that is specified by the opcode.

But, in practice, the computation of the N-operand aggregate function may begin as operands arrive at the AFU (before all operands arrive) and the AFU may perform the computation as a binary operation of a newly arrived operand and the accumulated result. With this reason, an expensive N-ary ALU may not be required.

4.3.4 AFU Allocation Table

The AFU allocation table keeps track of outstanding teams of threads that use the on-chip AFN for the collective operations. When a team of threads is assigned an afuID, the corresponding entry (indexed by the afuID) is allocated for the team of threads and stores appropriate information.

4.4 A Case for CMP-AFN

In this section, the analytical model for the barrier synchronization will be discussed.

4.4.1 Barrier Synchronization Latency for CMP-AFN

From each thread's perspective, the latency (T_{AFN_instr}) of AFN instructions can be defined as follows:

$$T_{AFN\_instr} = T_{Core\_to\_AFN} + T_{load\_imbalance} + T_{BPL} + T_{AFPL} + T_{AFN\_to\_Core}$$

For barrier synchronization, $T_{AFPL} = 0$. Therefore, when all threads start to execute the barrier instruction at the same time (i.e. $T_{load\_imbalance} = 0$), the latency for barrier synchronization will be as follows:

$$T_{barr\_cmpafn} = T_{Core\_to\_AFN} + T_{BPL} + T_{AFN\_to\_Core}$$

Table 4.2
A Case for CMP-AFN

Latencies and Configuration
CMP 4 cores
L1 64k+64k, 64B line
L1 latency 3 cycles
Shared L2 4 banks: 1 local and 3 remote
Shared L2 local/remote latency: 15/20 cycles
Write-back to read latency: 35-40 cycles
Transit time between core and AFN (one-way): 7 cycles
Barrier sync latency within AFN: 3 cycles
Reduction latency within AFN: 10 cycles
PutGet latency within AFN: 6 cycles

Table 4.3
GOMP latencies

T_{mutex_lock}	20	read mutex_var from L2
T_{mutex_unlock}	20	write-back mutex_var to L2
T_{++arr}	21	read arrived from L2 and increment
T_{--arr}	21	read arrived from L2 and decrement
T_{sem_post}	21	read sem_post from L2 and decrement
$T_{core2afn\_transit}$	7	transit time from core to AFN
$T_{afn2core\_transit}$	7	transit time from AFN to core

4.4.2 Barrier Synchronization Latency in a conventional CMP

`gomp_barrier_wait()` is the GNU implementation of barrier synchronization. As illustrated in Figure 4.5, it uses a shared variable `arrived`. Its execution consists of three stages– arrival stage, wake-up stage, and re-initialization stage. In the arrival stage, when a thread arrives at a barrier, it increments the `arrived` shared variable, then spin-wait using `gomp_sem_wait()`. When the last thread arrives at the barrier, it wakes-up the prior N-1 threads by calling `gomp_sem_post()` N-1 times (the wake-up stage). Then, all other threads atomically decrement `arrived` to re-initialize to zero for next barrier use (the re-initialization stage).

Bold faced functions in Figure 4.5 represent codes in the critical path of the latency. Therefore, the latency of `gomp_barrier_wait()` for N threads can be represented as follows:

$$T_{gomp\_barr} = \sum_{p=1}^{N}(T_{mutex\_lock} + T_{++arr} + T_{mutex\_unlock}) + \sum_{p=1}^{N} T_{sem\_post} + \sum_{p=1}^{N}(T_{mutex\_lock} + T_{--arr} + T_{mutex\_unlock})$$

Table 4.3 lists *minimum* time to take each function that are listed in the above latency equation. For example, the time for `gomp_mutex_lock()` will vary at run-

Table 4.4
barr sync latency comparison

	latencies
$T_{barr\_cmp\_afn}$	17 cycles
T_{gomp_barr}	572 cycles
$T_{gomp\_barr\_log}$	286 cycles

time due to lock contention and the Table shows the minimum time to execute the function.

`gomp_barrier_wait()` does not parallelize the arrival stage, as illustrated in Figure 4.5. In other words, even if all threads arrive at a barrier at the same time, `arrived` is incremented sequentially. If more than one shared variables are used, the arrival stage can be executed in parallel. In this case, the latency is as follows:

$$T_{gomp\_barr\_log} = \sum_{p=1}^{\log N}(T_{mutex\_lock} + T_{++arr} + T_{mutex\_unlock}) + \sum_{p=1}^{\log N} T_{sem\_post}$$

$$+ \sum_{p=1}^{\log N}(T_{mutex\_lock} + T_{--arr} + T_{mutex\_unlock})$$

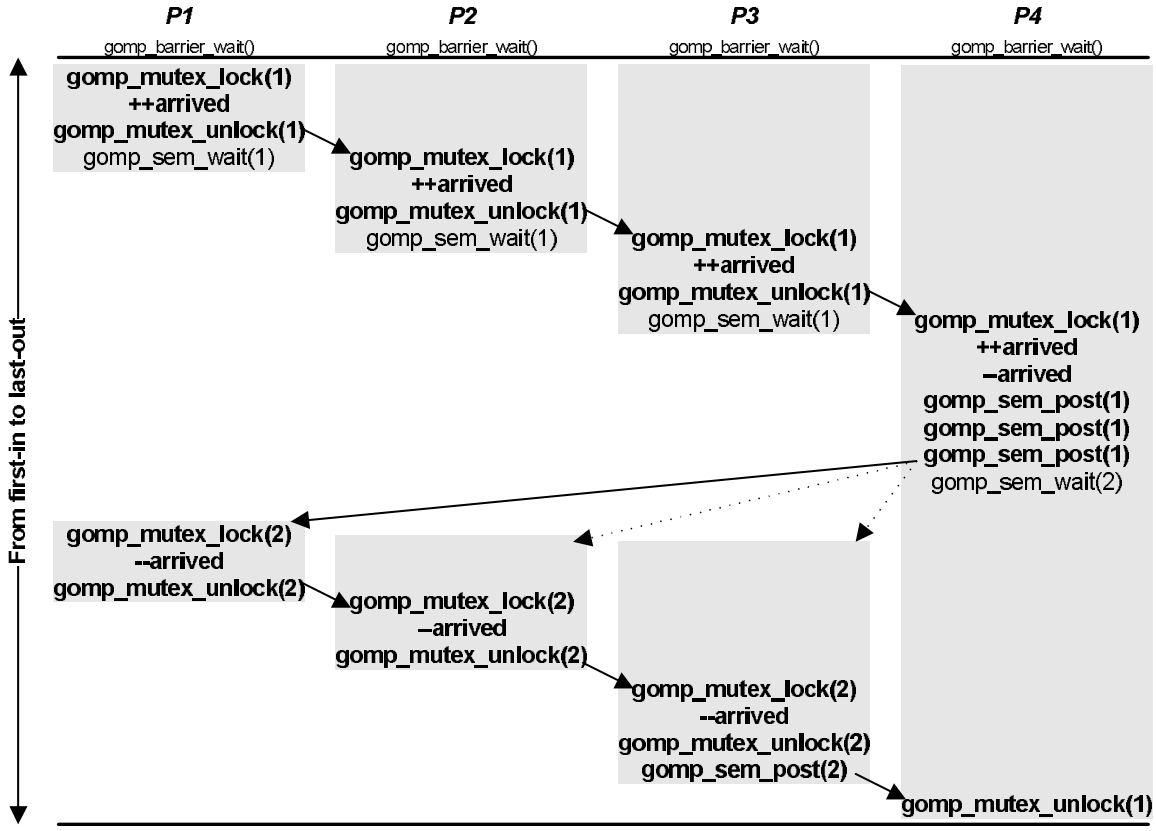


Fig. 4.5. gomp_barrier_wait() when all four threads arrived a barrier at the same time

5. EVALUATION METHODOLOGY AND RESULTS

The evaluation has been done using a cycle-accurate full-system simulation of a single chip shared-memory multi-core processor using *Simics* [43] and *GEMS* [44]. We evaluate CMP-AFN against CMP without on-chip AFN (CMP-REF), with various core counts. We use the EPCC OpenMP Microbenchmarks [7] and the SPEC OMP (OpenMP Benchmark Suite) [45–47] for the evaluation. We compiled and ran OpenMP benchmarks using GOMP (GNU OpenMP), an OpenMP implementation for GCC [5, 48].

5.1 Simulation Target Configurations

Simics is a full-system functional simulator that models various instruction set architectures and can execute commercial applications and operating systems. We created six Simics simulation targets – three CMP-REF (4, 8, and 16 cores) and three CMP-AFN targets (4, 8, and 16 cores). A 4-core CMP-AFN target is shown in Figure 5.1. All CMP-REF and CMP-AFN targets have Fedora Core 5 installed and contain IA32-based cores, 512 MB memory, and one 19GB IDE disk. Fedora Core 5 features Linux kernel 2.6.15 with SMP support and Native POSIX Thread Library (NPTL) version 2.4, which supports Futex (Fast Userspace Mutex) [4].

GEMS's Ruby enables the models of the cache memory hierarchy, the protocol controllers, and the on-chip interconnect in all simulation targets. It provides performance statistics for all cache and memory activities. Each target has a 32K-Byte private L1 cache for each core and 1 MByte-per-core shared L2 cache. These values are not intended to model a specific commercial chip, but to approximate the cache structure one might reasonably expect for a many-core design. The latest (64-bit extended) IA32-based 4-core Intel Core i7 and AMD Phenom II chips use 32K to

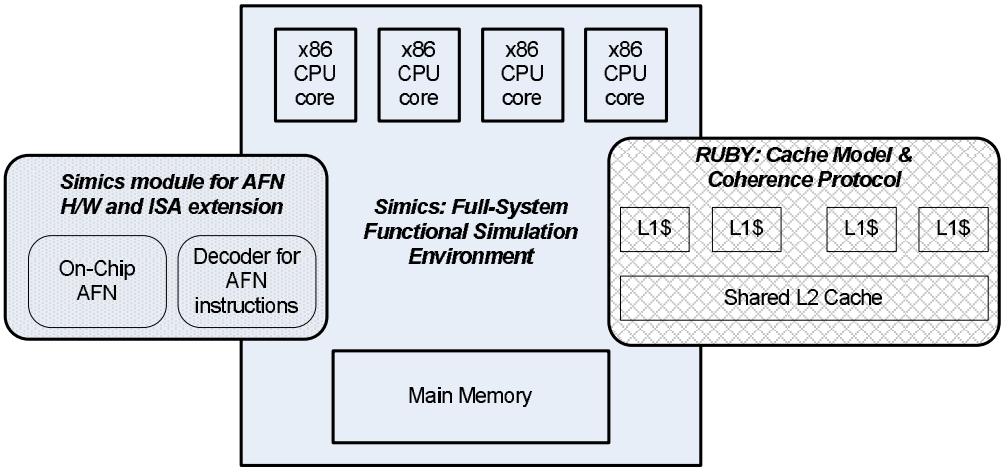


Fig. 5.1. A four-core CMP-AFN Simics Target

64K L1 caches. Our L2 cache design approximates the 6M to 8M shared L3 caches of those chips. We used Ruby's M̂OESI_CMP_directory for the on-chip cache coherence protocol. It is an on-chip two-level directory protocol using non-inclusive L1/L2 caching with blocking caches. This directory-based cache coherence protocol was selected because it scales better with higher core counts than a snoopy protocols.

For CMP-AFN targets, we developed a Simics module to implement on-chip AFN hardware logic and the instruction decoder for the AFN ISA extensions. We load the AFN instruction decoder module into the CMP-REF targets, as shown in Figure 5.1, to create CMP-AFN targets. The transit time between a core and an L2 cache bank is the one-way latency of a memory request traveling from the core to one of the L2 cache controllers through the on-chip interconnect. The same amount of time is used for the transit time between a core and the on-chip AFN. Table 5.1 shows system configurations for the target architectures. We used Simics version 3.0.29 and GEMS version 1.4. To use Ruby with the IA-32 architecture-based Simics targets, we created a Ruby patch for the IA-32 architecture for GEMS version 1.4, which has been integrated and released as part of GEMS version 2.0.

5.2 Re-targeting OpenMP Benchmarks for CMP-AFN

For the CMP-AFN architecture, synchronization primitives and thread management methods can be ported using AFN instructions to utilize the on-chip AFN for collective operations in the OpenMP applications. Because an OpenMP runtime library for the CMP-AFN does not exist, we replaced OpenMP constructs/routines in the OpenMP benchmark with the corresponding *AFN Library routines for OpenMP*. The AFN Library routines for OpenMP include AFN instructions and perform OpenMP barrier synchronization, lock, and reductions through the on-chip AFN in the target. Table 5.2 shows conversions from OpenMP constructs/routines to AFN library routines for OpenMP. Although we manually performed the conversion, this can also be done with an automatic source-to-source translator. We compiled the

Table 5.1
Configurations for Simulated Targets

CPU Cores	Intel Pentium 4, in-order 4-core, 8-core, and 16-core
Private L1 Cache	32KB 4-way, 64Byte-line L1 hit latency: 1-cycle
Shared L2 Cache	4-core CMPs: 4MB, 4 banks, 16-way 8-core CMPs: 8MB, 8 banks, 32-way 16-core CMPs: 16MB, 16 banks, 64-way
Cache Coherence Protocol	Directory Based Protocol i.e. Ruby's `MOESI_CMP_directory`
Main Memory	512MB total memory 200-cycle memory access
Platform	Tango (Fedora Core 5) Linux kernel 2.6.16 NPTL 2.4 (Native Pthreads Lib) with Futex GCC 4.2.1 with OpenMP support

Table 5.2
Replacing OpenMP constructs with AFN Library routines

OpenMP Constructs/Routines		AFN Lib Routines for OpenMP
`#pragma omp barrier`	$\Longrightarrow$	`afn_barrier_omp();`
`#pragma omp critical`		`afn_lock_omp();`
{ /* structured-block */ }	$\Longrightarrow$	{ /* structured-block */ }
		`afn_unlock_omp();`
`omp_init_lock(lock)`	$\Longrightarrow$	`afn_alloc_omp()`
`omp_destroy_lock(lock)`	$\Longrightarrow$	`afn_free_omp()`
`omp_set_lock(lock)`	$\Longrightarrow$	`afn_lock_omp()`
`omp_unset_lock(lock)`	$\Longrightarrow$	`afn_unlock_omp()`

OpenMP benchmarks with GCC 4.2 which features GOMP [48] (GNU OpenMP), an OpenMP implementation for GCC.

5.2.1 `barrier` Construct

The `barrier` construct specifies an explicit barrier at the point at which the construct appears. The syntax of the barrier construct for C/C++ is as follows:

```
#pragma omp barrier
```

When GCC 4.2 encounters `barrier` construct, it replaces `#pragma omp barrier` with `GOMP_barrier();`.

```
#pragma omp barrier  ⟹  GOMP_barrier();
```

For the CMP-AFN target, we replaced the `barrier` construct in the benchmarks with the afn_barrier_omp() routine before compile the benchmarks.

```
#pragma omp barrier  ⟹  afn_barrier_omp();
```

5.2.2 `critical` Construct

The `critical` construct restricts execution of the associated structured block to a single thread at a time. The syntax of the critical construct for C/C++ is as follows:

```
#pragma omp critical
{
    /* structured-block */
}
```

For a `critical` construct, GCC 4.2 places a pair of GOMP routines, `GOMP_critical_start()` and `GOMP_critical_end()`, immediately before and after the structured block, respectively.

```
#pragma omp critical          GOMP_critical_start();
{ /* structured-block */ } ⟹  { /* structured-block */ }
                              GOMP_critical_end();
```

For CMP-AFN target, we placed `afn_lock_omp()` and `afn_unlock_omp()` in the benchmarks immediately before and after the structured block, then compiled with GCC 4.2.

```
#pragma omp critical          afn_lock_omp();
{ /* structured-block */ } ⟹  { /* structured-block */ }
                              afn_unlock_omp();
```

5.2.3 OpenMP Locks

The OpenMP runtime library includes a set of general-purpose lock routines that can be used for synchronization. These general-purpose lock routines operate on OpenMP locks represented by OpenMP lock variables. An OpenMP lock variable must only be accessed through the routines described in this section.

An OpenMP lock may be in one of the following states: uninitialized, unlocked, or locked. If a lock is in the unlocked state, a thread may set the lock, which changes

its state to locked. The thread which sets the lock is then said to own the lock. A thread which owns a lock may unset that lock, returning it to the unlocked state. A thread may not set or unset a lock which is owned by another thread.

The simple lock routines are as follows:

- `omp_init_lock()` routine initializes a simple lock.
- `omp_destroy_lock()` routine uninitializes a simple lock.
- `omp_set_lock()` routine waits until a simple lock is available, and then sets it.
- `omp_unset_lock()` routine unsets a simple lock.

GCC 4.2 replaces OpenMP lock routines with corresponding Pthreads routines as follows:

```
omp_init_lock(lock)     ⟹  pthread_mutex_init (lock, NULL)
omp_destroy_lock(lock)  ⟹  pthread_mutex_destroy (lock)
omp_set_lock(lock)      ⟹  pthread_mutex_lock (lock)
omp_unset_lock(lock)    ⟹  pthread_mutex_unlock (lock)
```

For CMP-AFN targets, we manually replaced OpenMP lock routines with the AFN routines as follows:

```
omp_init_lock(lock)     ⟹  afn_alloc_omp()
omp_destroy_lock(lock)  ⟹  afn_free_omp()
omp_set_lock(lock)      ⟹  afn_lock_omp()
omp_unset_lock(lock)    ⟹  afn_unlock_omp()
```

5.3 Compiling OpenMP Benchmarks for CMP Simics Targets: Checkpointing

Figure 5.2 shows how we create executables for the simulated targets. The first step is to identify the region of code in each benchmark that we want to run and measure the performance. In Simics, for each simulated processor architecture, a

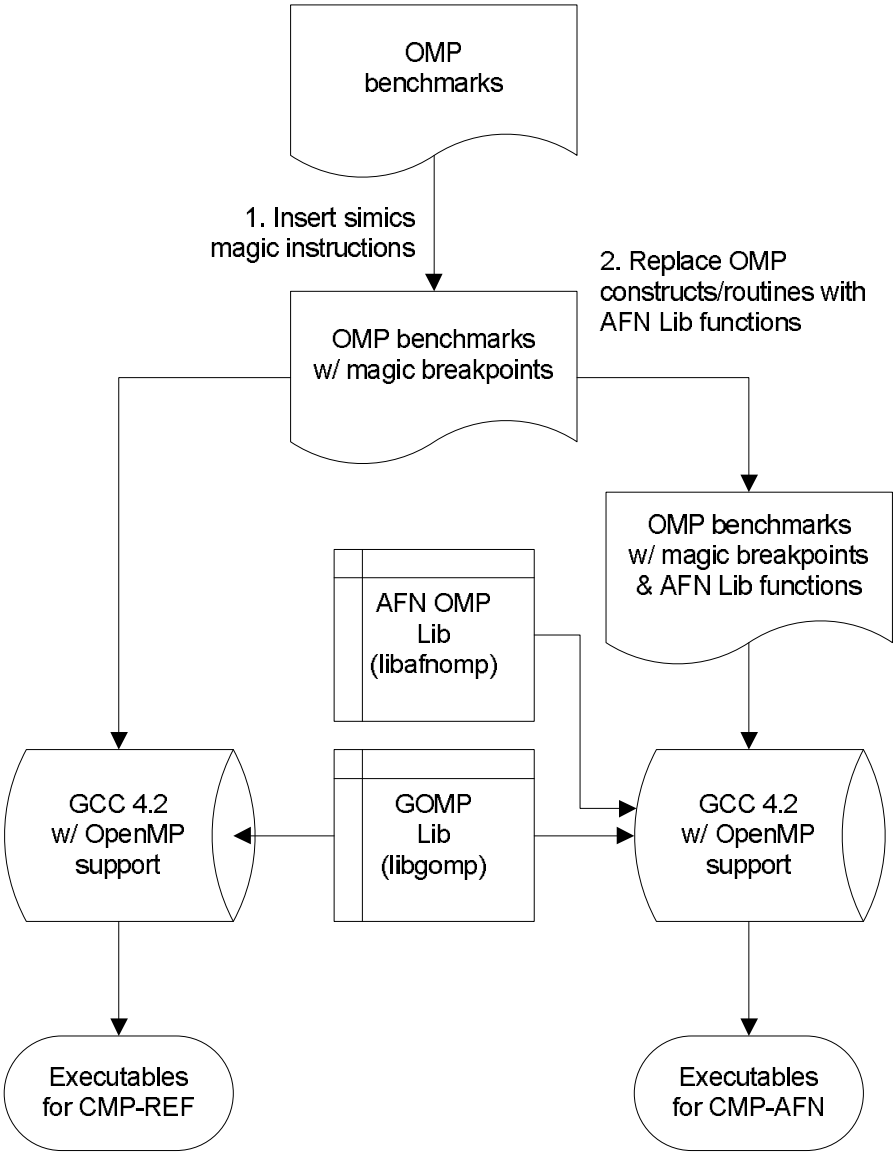


Fig. 5.2. Steps to create executables for CMP-AFN and CMP-REF targets

special no-operation instruction has been chosen to be a *magic instruction* for the simulator. For x86, it is `xchg %bx, %bx`. When a magic breakpoint is triggered by the execution of the magic instruction, the simulator stops and returns to prompt. To create two breakpoints, we insert two magic instructions immediately before and after the identified chunk of source code.

The benchmark is run without the cache memory model up to the first breakpoint. At the first magic breakpoint, we checkpoint the simulation by saving the entire state of the simulation to disk. We resume simulation with Ruby enabled to model cache memory and the coherence protocol, and we start to collect performance data. When the simulation stops again at the second breakpoint, the performance statistics are dumped for analysis.

After inserting magic instructions and replacing OpenMP constructs with the AFN Lib routines, we cross-compile the benchmark source code on the host machine where GCC 4.2 is installed. Using SimicsFS, we mount the host file system from the simulated target so that the simulated target can copy the cross-compiled executables from the host and can access the GCC 4.2 run-time library to run the executables.

The following GCC compiler option was used:

```
-fopenmp -march=i386 -O3
```

5.4 Overview of OpenMP Benchmarks

We ran the EPCC OpenMP Microbenchmarks [7], which provide microbenchmark information for the OpenMP `barrier` construct, OpenMP lock runtime library routines, and `reduction` constructs among other constructs. We also ran the SPEC OMP (OpenMP Benchmark Suite) [46,47]. Table 5.4 shows static counts of OpenMP constructs and run-time routines in the SPEC OMP benchmarks that are executed on the simulation targets for the evaluation. Because of the simulation speed and the small memory capacity of the simulated targets, we ran these benchmarks with a small data set and with reduced iteration counts.

```
main() {
    read_input();
#pragma omp single
    {
        __asm__ __volatile__ ("xchg %bx, %bx"); // breakpoint #1
    }
    "code to be measured..."
#pragma omp barrier
    __asm__ __volatile__ ("xchg %bx, %bx"); // breakpoint #2
    "other program code..."
}
```

Fig. 5.3. Structure of a Program with magic instructions

Table 5.3
Static Counts of OpenMP Constructs and Run-Time Routines in SPEC OMP Benchmarks for Evaluation

	`barrier`	`critical`	`OpenMP locks`	`reduction`	implicit barrier
316.applu_m	2				
320.equake_m					2
324.apsi_m			1		
330.art_m		1			
332.ammp_m			3		

In the SPEC OMP benchmarks, **316.applu_m** contains an *explicit* OpenMP `barrier` construct for the barrier synchronization. **320.equake_m** does not contain any OpenMP constructs (except parallel loop constructs) that are relevant to CMP-AFN. Implicit barriers in 320.equake_m have been replaced with an AFN barrier synchronization since there is an implicit barrier at the end of the `parallel` construct in OpenMP [49]. **330.art_m** contains the `critical` OpenMP construct that restricts execution of the associated structured block to a single thread at a time. For a `critical` construct, GCC/GOMP places `pthread_mutex_lock()` and `pthread_mutex_unlock()` at the beginning and the end of the associated block, respectively. In CMP-AFN, AFNLOCK and AFNUNLOCK are used for the critical construct. **332.ammp_m** contains OpenMP locks that restricts execution of the associated structured block to a single thread at a time. In CMP-AFN, AFNLOCK and AFNUNLOCK are used for the `omp_set_lock()` and `omp_unset_lock()`, respectively.

5.5 EPCC Microbenchmark

We ran the EPCC OpenMP Microbenchmark suite [7] [50] to evaluate the performance of collective operations in the proposed architecture, compared to the CMP reference architecture. Due to the simulation speed and the small memory capacity of the simulated targets, we ran the benchmark with the reduced iteration counts.

```
void delay(int delaylength) {
  int  i;
  float a=0.;

  for (i=0; i<delaylength; i++) a+=i;
  if (a < 0) printf("%f \n",a);
}
```

Fig. 5.4. `delay()`: EPCC microbenchmark

The Microbenchmarks includes `syncbench.c`, which consists of `testbar()`, `testlock()`, and `testred()` functions to provide microbenchmark information for barrier synchronization construct, OMP locks runtime library routines, and reduction constructs, respectively. We ran a modified version of these functions with the slightly smaller loop iteration counts as shown below, in order to reduce the Simics simulation time. `delay(delaylength)`, shown in Figure 5.4, is called in each inner loop body for all microbenchmarks.

5.5.1 syncbench.c: testbar() for barrier synchronization

As shown in Figure 5.5, `testbar()` includes nested loops where the innerloop body contains barrier synchronization. We used the `CMP_AFN` macro to keep the same source for both CMP-REF and CMP-AFN with conditional compilation. For CMP-AFN, each thread calls the `afn_barrier_omp()` routine. For CMP-REF targets, threads perform barrier synchronization via GOMP's barrier routine which uses the shared memory. We measured the performance of the `testbarr()`.

5.5.2 syncbench.c: testlock() for OpenMP locks

As shown in Figure 5.6, `testlock()` includes nested loops, where the innerloop body contains a pair of OpenMP lock and unlock routines. We used `CMP_AFN`

```
void testbar() {
  int j,k;
  unsigned int rank;
  for (k=0; k<=OUTERREPS; k++){
#pragma omp parallel private(j, rank)
    {
      rank = omp_get_thread_num();
      for (j=0; j<innerreps; j++) {
        delay(delaylength);
#ifdef CMP_AFN
   afn_barrier_omp(rank);
#else
#pragma omp barrier
#endif
      }
    }
  }
}
```

Fig. 5.5. `testbar()`: EPCC microbenchmark for barrier synchronization

macro definition to keep the same source for both CMP-REF and CMP-AFN with conditional compilation. For CMP-AFN, each thread calls `afn_lock_omp()` and `afn_unlock_omp()` routines. For CMP-REF, threads perform OpenMP locks via Pthreads' mutex/unmutex routines which use shared memory. We measured the performance of the `testlock()`.

5.5.3 syncbench.c: testred() for reduction

As shown in Figure 5.7, `testred()` includes nested loops where the innerloop performs reduction add at the end of the loop. We used `CMP_AFN` macro to keep the same source for both CMP-REF and CMP-AFN with conditional compilation. For CMP-AFN, each thread calls the `afn_add_omp()` routine. For CMP-REF targets, threads perform the reduction add via the shared memory. We measured the performance of the `testred()`.

5.6 SPEC OMP2001 Benchmark Suite

We ran the SPEC OMP 2000 Benchmark suite [47] [46] to evaluate the performance of the proposed architecture, ISA extensions, and the AFN Library for OpenMP against the CMP reference architecture. Due to the simulation speed and the small memory capacity of the simulated targets, we run the SPEC OMP Benchmark with small data set and with the reduced iteration counts.

Out of 11 SPEC OMP Benchmarks, we selected four benchmarks–316.applu_m, 320.equake_m, 330.art_m, and 332.ammp_m–for both `barrier` construct (explicit), and implicit barrier synchronization (due to `omp parallel for` constructs) , `critical` construct, and OpenMP lib routines for Locks/Unlocks, respectively.

We had gfortran compilation errors for 314.mgrid_m and 318.galgel_m due to not being compliant to OpenMP spec 2.5 and unspecified errors, respectively. With the small data set, 328.fma3d_m did not reach the `critical` OpenMP constructs. With the medium data set, 328.fma3d_d is too long to simulate in Simics environment.

```
void testlock() {
  int j,k;
  unsigned int rank;
  omp_lock_t lock;
#ifndef CMP_AFN
  omp_init_lock(&lock);
#endif
  for (k=0; k<=OUTERREPS; k++) {
#pragma omp parallel private(j,rank)
    {
      rank = omp_get_thread_num();
      for (j=0; j<innerreps/nthreads; j++) {
#ifdef CMP_AFN
        afn_lock_omp(rank);
#else
        omp_set_lock(&lock);
#endif
        delay(delaylength);
#ifdef CMP_AFN
        afn_unlock_omp(rank);
#else
        omp_unset_lock(&lock);
#endif
      }
    }
  }
}
```

Fig. 5.6. `testlock()`: EPCC microbenchmark for OpenMP Lock. OUTERREPS=10, innerreps=128, and delaylength=500.

```
void testred() {
  int j,k, mySum, *gSum;
  unsigned int rank;
  gSum = &mySum;
  for (k=0; k<=OUTERREPS; k++) {
    mySum = k;
#pragma omp parallel for private(rank) reduction(+:mySum)
    for (j=0; j<innerreps; j++) {
      rank = omp_get_thread_num();
      mySum += j;
      delay(delaylength);
#ifdef CMP_AFN
      afn_add_omp(rank, mySum, gSum);
#endif
    }
  }
}
```

Fig. 5.7. `testred()`: EPCC microbenchmark for reduction

Table 5.4
OpenMP constructs used in SPEC OMPM2001 benchmark suite: `BARR` indicates for `BARRIER` construct, `CRIT` for `CRITICAL` construct, `LOCK` for OpenMP locks, `RED` for `REDUCTION` clause, and `i.barr` for implicit barrier synchronization. The numbers are static counts.

	BARR	CRIT	LOCK	RED	i.barr
310.wupwise_m		1		2	
312.swim_m				1	
314.mgrid_m				1	
316.applu_m	2			5	
318.galgel_m					
320.equake_m					2
324.apsi_m				1	
326.gafort_m			400000	3	
328.fma3d_m		1		8	
330.art_m		1			
332.ammp_m			3		

Table 5.5
SPEC OMPM2001 Benchmark Suite Description

	Application area	Lang	#files	Status
310.wupwise_m	QCD	F77	25	test FAILED ("killed")
312.swim_m	Shallow water	F77	1	Simics decoder issue
314.mgrid_m	Multigrid solver	F77	1	Not compliant to OMP2.5
316.applu_m	Fluid dynamics	F77	20	**pass**
318.galgel_m	Fluid dynamics	f90	39	gfortran compilation errors
320.equake_m	Earthquake model	C	1	**pass**
324.apsi_m	Air pollution	F77	1	Simics decoder issue
326.gafort_m	Genetic algorithm	f90	1	test FAILED ("killed")
328.fma3d_m	Crash simulation	f90	101	too long to simulate
330.art_m	Image Recognition	C	1	**pass**
332.ammp_m	Computational Chem	C	31	**pass**

```
1  !$omp parallel
2  !$omp&  default (shared)
3  !$omp&  private (i0, i1, ipx, ipy, j0, j1, k, l, mt, nt, npx, npy)
4  !$omp&  shared (nx, ny, nz, omega)
5  c---------------------------------------------------------------------
6  c Partition X-Y plane amongst processors in two dimensions.
7  c---------------------------------------------------------------------
8        nt = 1
9        mt = 0
10       npx = int (sqrt (dble (nt)))
11
12       DO WHILE ((npx .gt. 1) .and. (mod (nt, npx) .ne. 0))
13          npx = npx - 1
14       END DO
15
16       npy = nt / npx
17       ipx = mod (mt, npx)
18       ipy = mt / npx
19       i0 = (ipx + 0) * (nx - 2) / npx + 2
20       i1 = (ipx + 1) * (nx - 2) / npx + 1
21       j0 = (ipy + 0) * (ny - 2) / npy + 2
22       j1 = (ipy + 1) * (ny - 2) / npy + 1
23       DO l = 2, npx + npy + nz - 3
24          k = l - ipx - ipy
25          IF ((1 .lt. k) .and. (k .lt. nz)) THEN
26 c-----------------------------------------------------------------------
27 c        form the lower triangular part of the jacobian matrix
28 c-----------------------------------------------------------------------
29             CALL jacld (i0, i1, j0, j1, k)
30 c-----------------------------------------------------------------------
31 c        perform the lower triangular solution
32 c-----------------------------------------------------------------------
33             CALL blts( isiz1, isiz2, isiz3,
34      >           nx, ny, nz, i0, i1, j0, j1, k,
35      >           omega, rsd, tv, a, b, c, d )
36          END IF
37 !$omp barrier
38       END DO
39
40       DO l = npx + npy + nz - 3, 2, -1
41          k = l - ipx - ipy
42          IF ((1 .lt. k) .and. (k .lt. nz)) THEN
43 c-----------------------------------------------------------------------
44 c        form the strictly upper triangular part of the jacobian matrix
45 c-----------------------------------------------------------------------
46             CALL jacu(i0, i1, j0, j1, k)
47 c-----------------------------------------------------------------------
48 c        perform the upper triangular solution
49 c-----------------------------------------------------------------------
50             CALL buts( isiz1, isiz2, isiz3,
51      >           nx, ny, nz, i0, i1, j0, j1, k,
52      >           omega, rsd, tvu, du, au, bu, cu )
53          END IF
54 !$omp barrier
55       END DO
56 !$omp end parallel
```

Fig. 5.8. ssor.f from SPEC OMP 316.applu_m

5.6.1 316.applu_m

316.applu_m contains OpenMP `barrier` construct for the barrier synchronization. Figure 5.8 shows an OpenMP `parallel` region (ssor.f 138-209) that contains two `DO` statements with `barrier` at the end of each loop body. This parallel region contributes more than 80% of total execution time [8].

5.6.2 320.equake_m

320.equake_m is a benchmark in SPEC OMP where there is no OpenMP construct (except parallel loop constructs) that are relevant to CMP-AFN. For `omp parallel for` construct, GCC places `GOMP_barrier()` (barrier synchronization) at the end of the loop. We call them implicit barrier synchronization because they are not explicitly specified by the programmer, but implicitly placed by the compiler because OpenMP specification requires. In 320.equake_m, implicit barriers have been replaced with AFN barrier sync.

5.6.3 324.apsi_m

324.apsi_m contains a reduction clause in one of parallel loops. Currently, this benchmark fails due to one of outstanding Simics bugs (decode issue).

5.6.4 330.art_m

330.art_m contains the `CRITICAL` OpenMP construct that restricts execution of the associated structured block to a single thread at a time. For a `critical` construct, GOMP puts `pthread_mutex_lock` and `pthread_mutex_unlock()` at the beginning and the end of the associated block, respectively. In CMP-AFN, AFNLOCK and AFNUNLOCK are used for the critical construct.

5.6.5 332.ammp_m

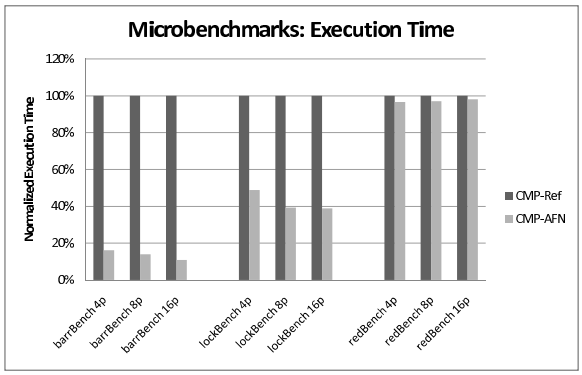
332.ammp_m contains OpenMP locks that restricts execution of the associated structured block to a single thread at a time. In CMP-AFN, AFNLOCK and AFNUNLOCK are used for the omp_set_lock() and omp_unset_lock(), respectively.

5.7 Performance Evaluation

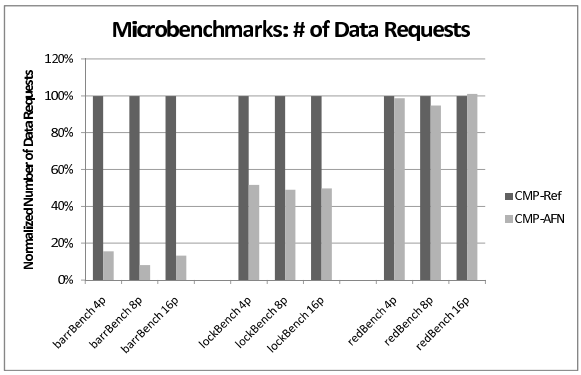
The potential performance improvement in CMP-AFNs comes from reductions in memory accesses, coherence protocol messages, and a dynamic instruction count. Because the on-chip AFN performs the collective operations, fewer memory accesses and coherence messages are required. The performance discussed in this section is based on busy-waiting based AFN synchronization. A blocking AFN synchronization mechanism such as this reduces the number of instructions that are executed for the synchronization primitives as well as network traffic between cores and the on-chip AFN.

Figures 5.9 and 5.10 summarize the performance results from EPCC OpenMP Microbenchmarks and SPEC OMP Benchmark, respectively. Results are presented in execution times (Figures 5.9(a) and 5.10(a)), number of memory accesses (Figures 5.9(b) and 5.10(b)), number of L1 DCache misses (Figures 5.9(c) and 5.10(c)), and number of L2 Cache misses (Figures 5.9(d) and 5.10(d)). The CMP-AFN performance numbers are normalized to those of CMP-REF with the same number of cores.

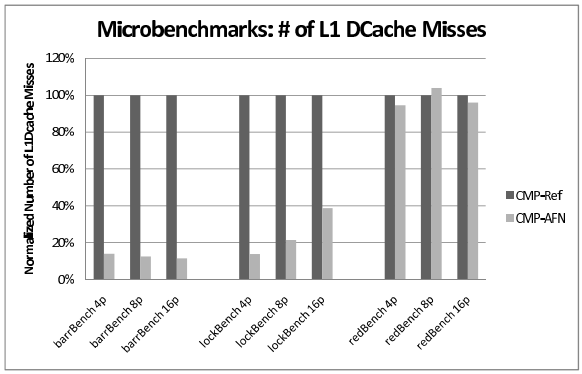
Overall, as shown in Figures 5.9 and 5.10, CMP-AFN outperforms CMP-REF in terms of execution time and shows even more performance benefits as the core count increases from 4 to 8 to 16 cores. The on-chip AFN significantly improves the performance of barrier synchronization and reduces the number of L1 DCache misses by more than 50% in 316.applu_m with 16 cores. The reduced number of L1 DCache misses results mostly from the absence of invalidation messages on the on-chip network. CMP-AFN reduces the number of memory accesses as shown in



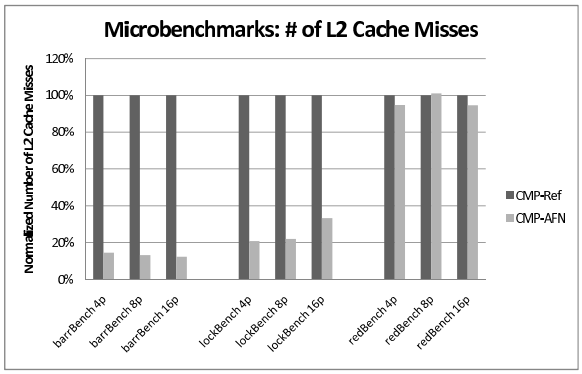
(a) Execution Time



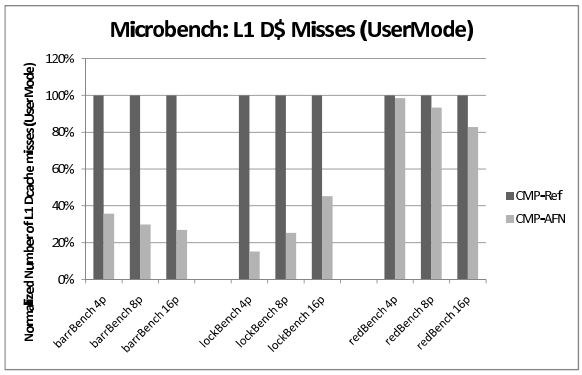
(b) Number of memory accesses



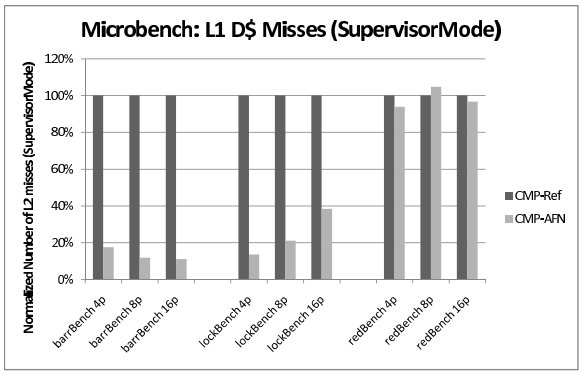
(c) Total number of L1 DCache misses in the system



(d) Total number of L2 cache misses in the system



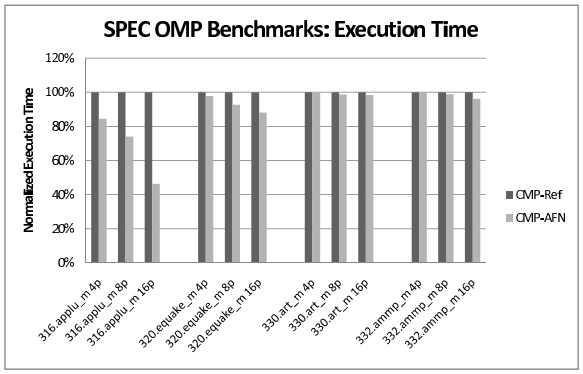
(e) L1 DCache Misses (UserMode)



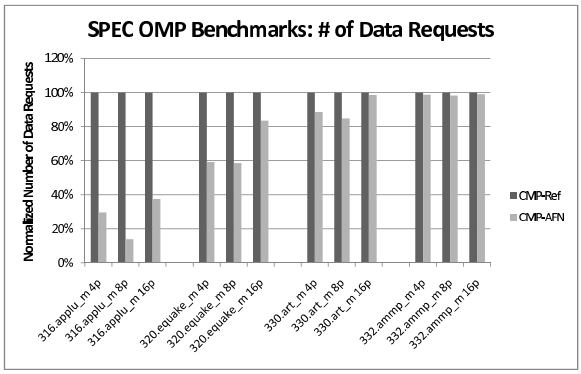
(f) L1 DCache Misses (SupervisorMode)

Fig. 5.9. Summary of EPCC OpenMP Microbenchmark Results

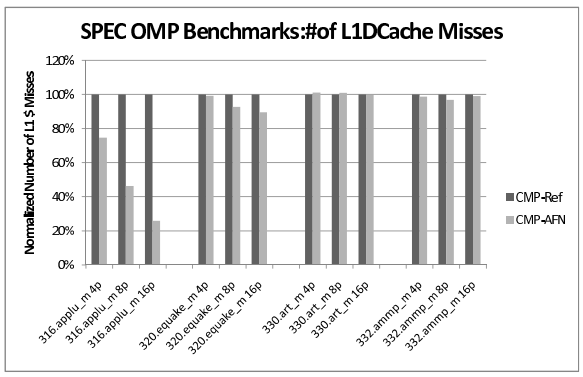
Figures 5.9(b) and 5.10(b), which indicates a reduced number of memory instructions are executed.



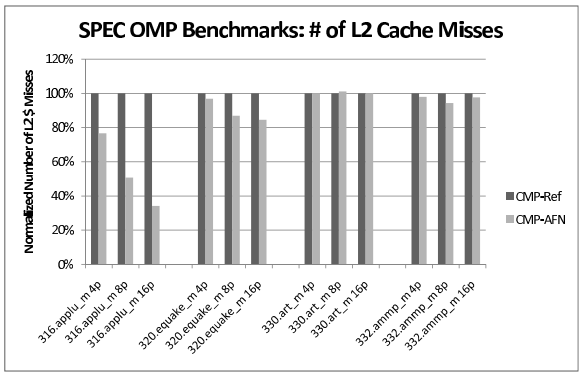
(a) Execution Time



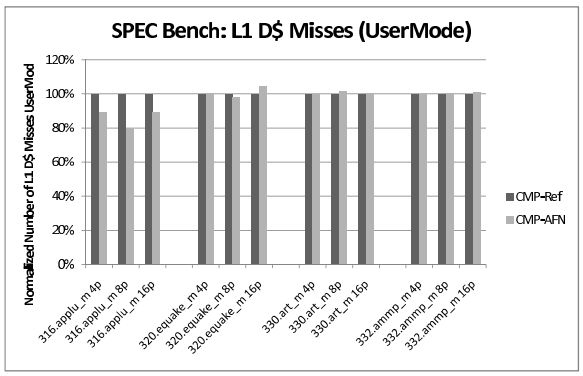
(b) Number of memory accesses



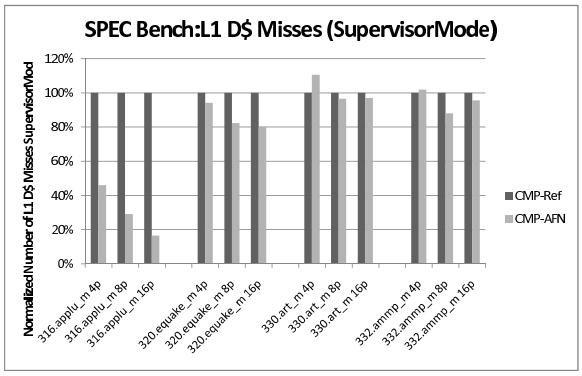
(c) Total number of L1 DCache misses in the system



(d) Total number of L2 Cache misses in the system



(e) L1 DCache Misses (UserMode)



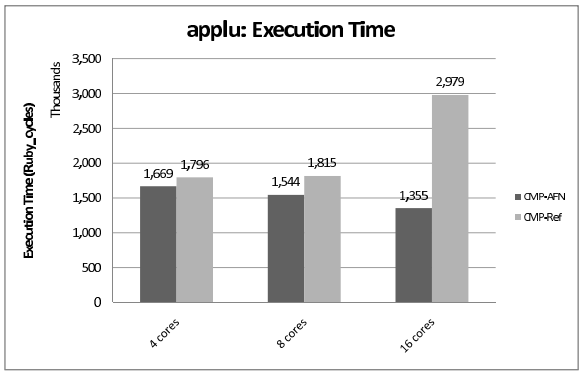
(f) L1 DCache Misses (SupervisorMode)

Fig. 5.10. Summary of SPEC OMP Benchmark Results

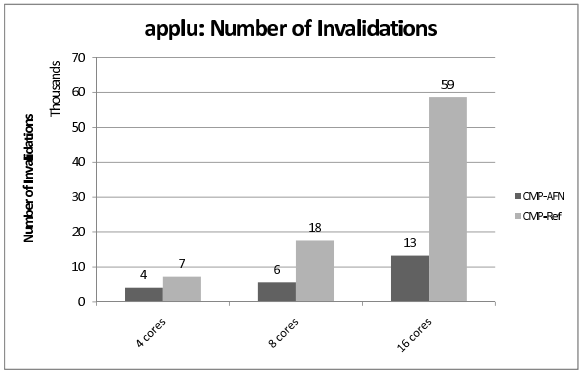
5.7.1 Performance Evaluation of 316.applu_m

In this section, we discuss the performance results of **applu**, which gets substantial benefits from the on-chip AFN, as shown in Figure 5.11(a).

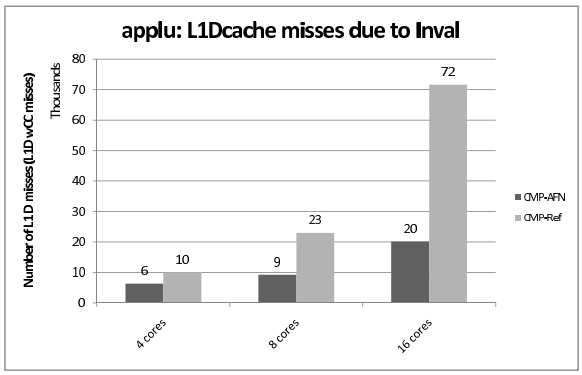
Memory references due to barrier synchronization cause significant a number of cache line invalidations in CMP-REF, compared to CMP-AFN, as shown in Figure 5.11(b). Cache line invalidations cause L1 data cache misses while threads are performing `mutex` on shared variables. Figure 5.11(c) shows the number of L1D cache misses *due to* cache line invalidations. And Figure 5.11(d) shows the per-thread penalty (accumulated latency) for L1D cache misses due to cache line invalidations.



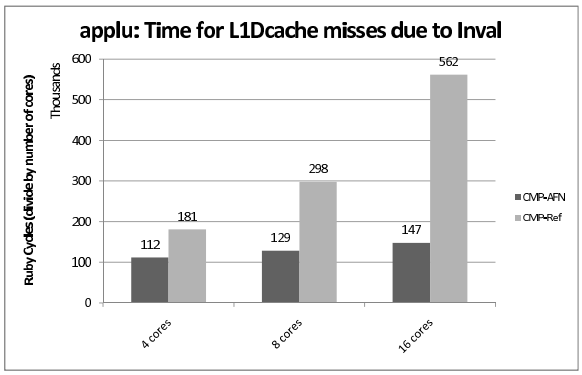
(a) Execution Time



(b) Total number of invalidations



(c) L1 DCache misses due to cache line invalidation



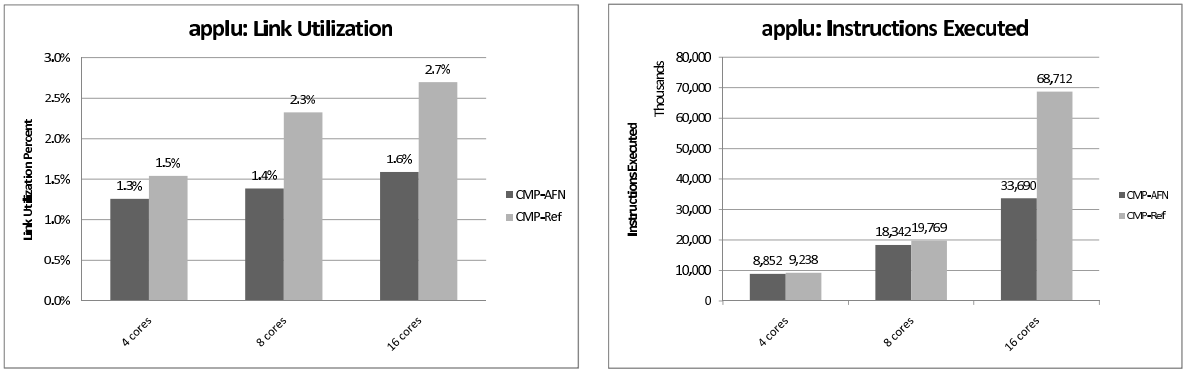
(d) Per-thread penalty for L1D misses due to cache line invalidations

Fig. 5.11. SPEC OMP 316.applu_m

In CMP-REF, the per-thread penalty for L1D cache misses due to invalidations increases as the thread count increases. On the other hand, in CMP-AFN, the per-thread penalty for L1D misses due to invalidations does not increase significantly. The penalty for L1D misses due to invalidations accounts for 10%, 16%, and 19% of the total execution time for 4-, 8-, and 16-core CMP-REF; and 7%, 8%, and 11% for CMP-AFN.

Because in CMP-AFN threads perform barrier synchronization via on-chip AFN, not via `mutex` on shared variables, the performance data show that the number of cache line invalidations and the number of L1 data cache misses due to invalidations are reduced significantly, compared to CMP-REF. As higher thread counts causes more cache line invalidation and L1D cache misses in CMP-REF due to higher degree of contention on shared synchronization variables, the benefit of the on-chip AFN gets more substantial as we get more threads in the systems as shown in Figure 5.11(a).

Figure 5.12(a) shows the network link utilization percentage. The link utilization indicates the network traffic throughout the simulation. Due to the higher number of invalidations and cache misses, all CMP-REF targets have a higher link utilization percentage than CMP-AFN counterpart. CMP-REF targets executed slightly more



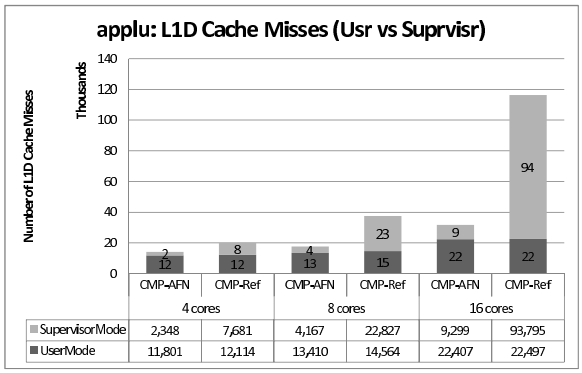
(a) Link utilization percent

(b) Instruction Counts

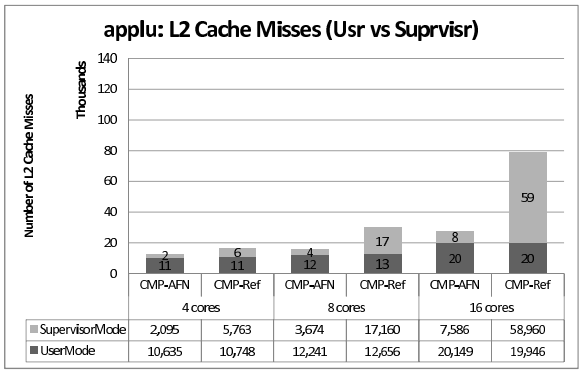
Fig. 5.12. SPEC OMP 316.applu_m (continued)

instructions than CMP-AFN for 4- and 8-cores. For 16-core CMP-REF, significantly more instructions are executed as shown in Figure 5.12(b).

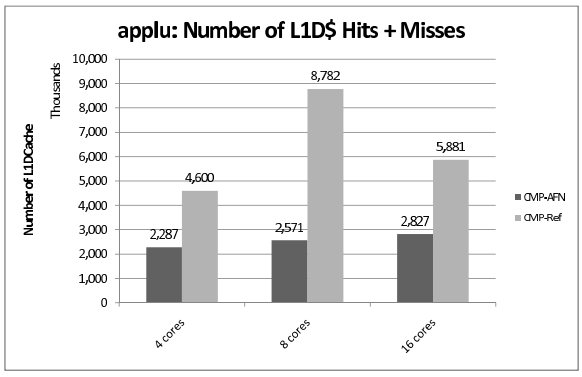
The breakdown of L1 data cache misses into user and supervisor mode (Figure 5.13(a)) shows that in CMP-REF the number of cache misses in supervisor mode accounts for significant fraction of the total number of L1 data cache misses due to FUTEX (Fast Userlevel muTEX) where the kernel handles the mutex variables [5]. This indicates that on-chip AFN allows CMP-AFNs to spend less time on performing `mutex` and execute less instructions in synchronization primitives than CMP-REFs.



(a) Total number of L1 DCache misses by Mod (User vs. Supervisor)



(b) Total number of L2 cache misses by Mode (User vs. Supervisor)



(c) L1D misses plus hits

Fig. 5.13. SPEC OMP 316.applu_m (continued)

Breakdown of Execution Time

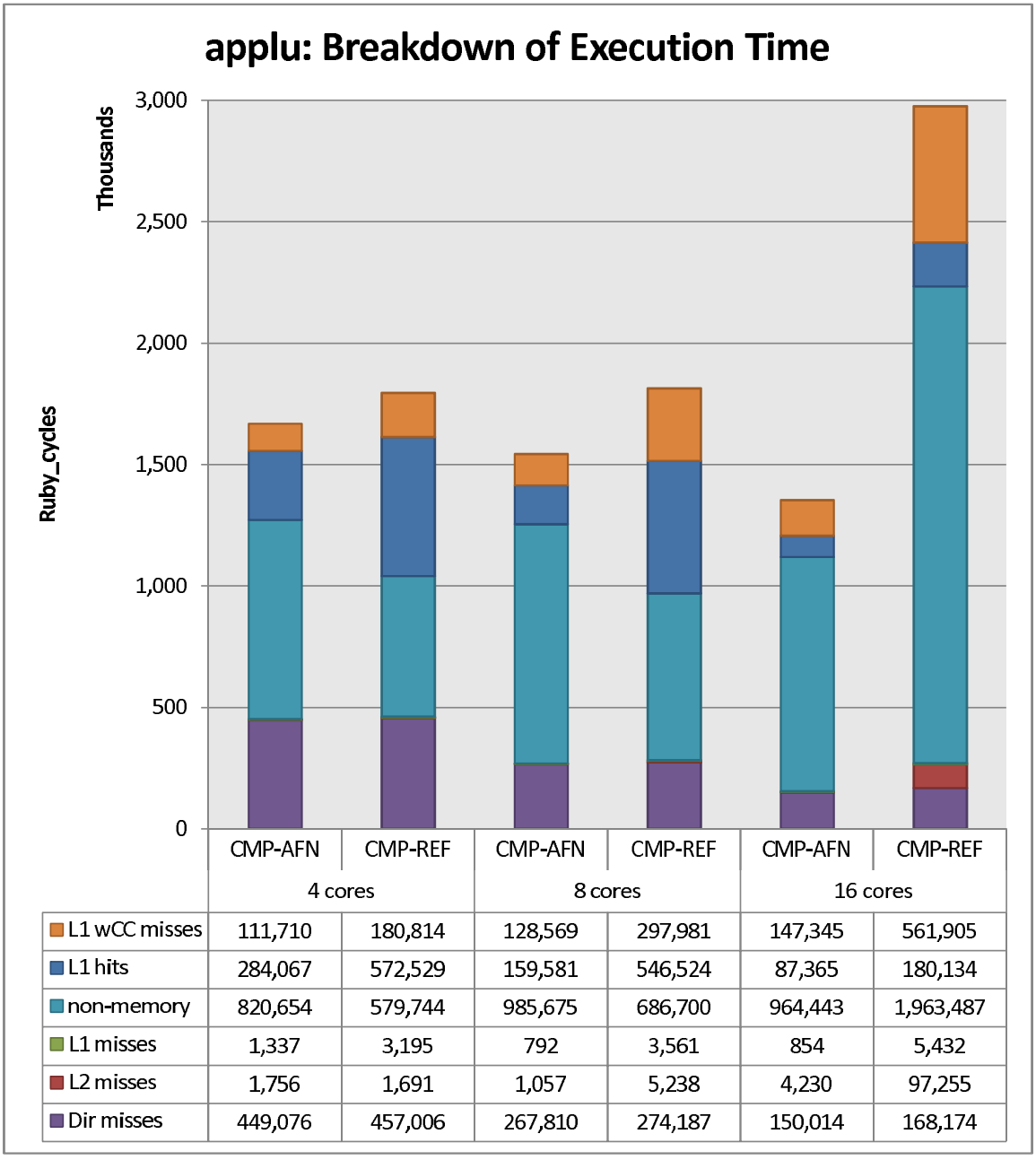
Figure 5.14 shows the breakdown of the execution time (cycles). For each simulation, the total number of cycles to complete the benchmark is broken down into times that are needed for (1) non-memory instructions, (2) L1 hits, (3) L1 data cache misses with cache coherence activity [L1 wCC misses], (4) L1 misses without cache coherent activity [L1 misses], (5) L2 misses, and (6) Directory misses.

CMP-AFNs spend less time on *L1 wCC misses* than CMP-REFs as the barrier synchronization took place without spin-waiting on shared flags, which cause excessive coherence traffic (invalidations), causing L1 wCC misses (as we discussed earlier as the penalty for L1D misses due to cache line invalidations). For the same reason, CMP-REFs spend more time on L1 hits than CMP-AFNs as it spin-waits on the shared flags. For CMP-AFN, the number of non-memory instructions includes AFN instructions and `nop` instructions that are used to implement the latency of on-chip AFN instructions. The number of cycles that is required for L1 misses and L2 misses does not account much in the total execution time and can be ignored in the discussion here. CMP-AFN and CMP-REF spend similar time on Directory misses.

In summary, for 4- and 8-core CMPs, *L1 wCC misses* accounts for performance difference between two CMP architectures as the time for all others are similar.

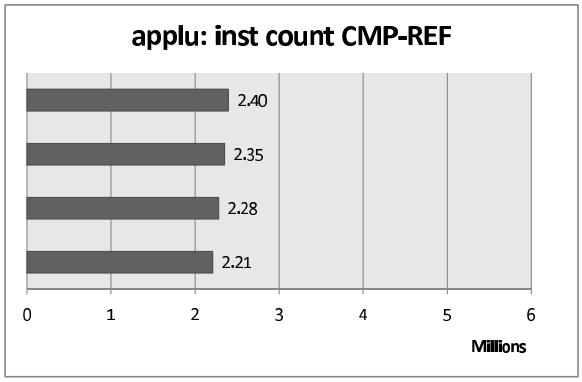
Distribution of Instruction Counts

Instruction count per core may indicate whether load-balance within a team of threads although no other information in the simulation stats can provide the per-core data. As shown in Figure 5.15, the number of instructions among threads is evenly distributed with low core count.

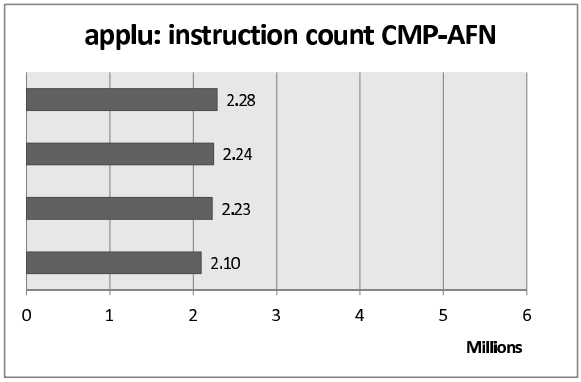


	CMP-AFN	CMP-REF	CMP-AFN	CMP-REF	CMP-AFN	CMP-REF
	4 cores		8 cores		16 cores	
L1 wCC misses	111,710	180,814	128,569	297,981	147,345	561,905
L1 hits	284,067	572,529	159,581	546,524	87,365	180,134
non-memory	820,654	579,744	985,675	686,700	964,443	1,963,487
L1 misses	1,337	3,195	792	3,561	854	5,432
L2 misses	1,756	1,691	1,057	5,238	4,230	97,255
Dir misses	449,076	457,006	267,810	274,187	150,014	168,174

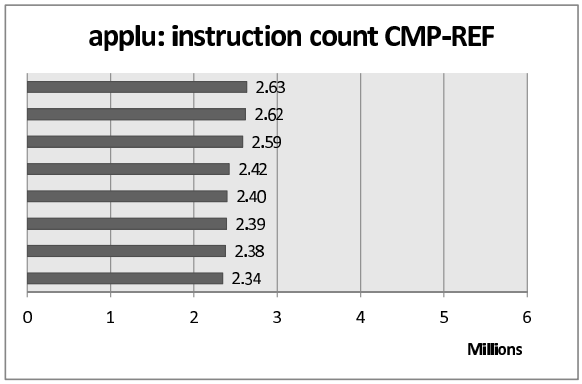
Fig. 5.14. applu: Breakdown of Execution Time by Cache Misses



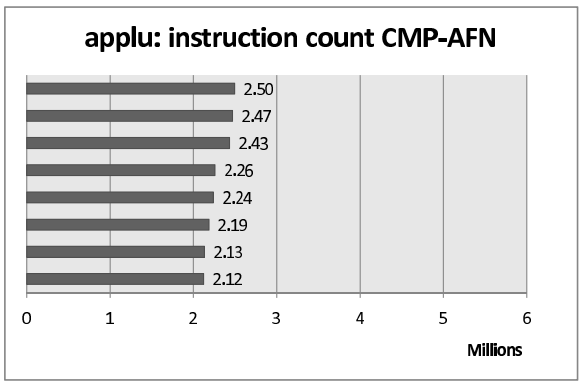
(a) 4-core CMP-REF



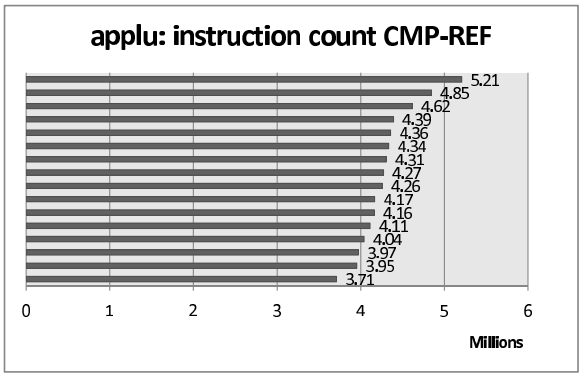
(b) 4-core CMP-AFN



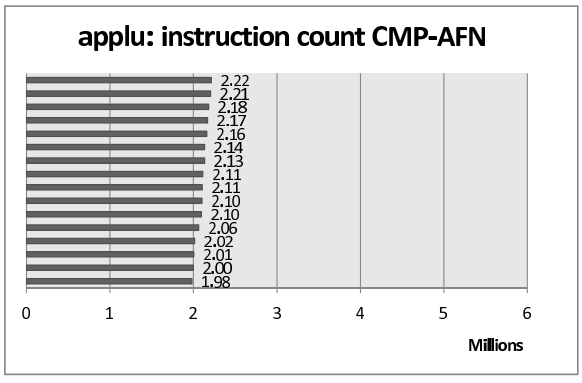
(c) 8-core CMP-REF



(d) 8-core CMP-AFN

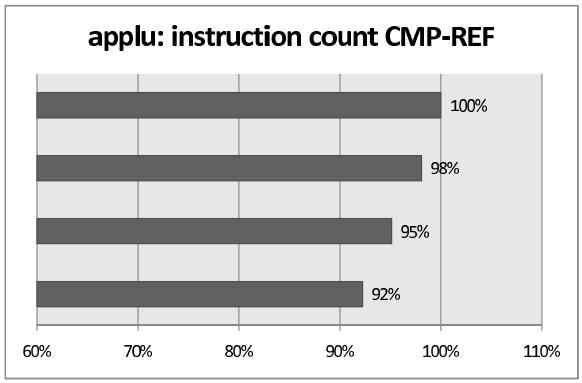


(e) 16-core CMP-REF

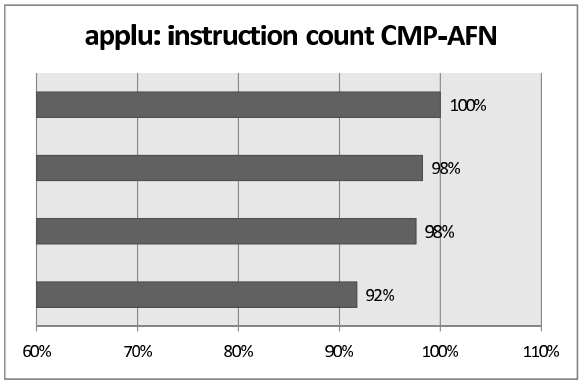


(f) 16-core CMP-AFN

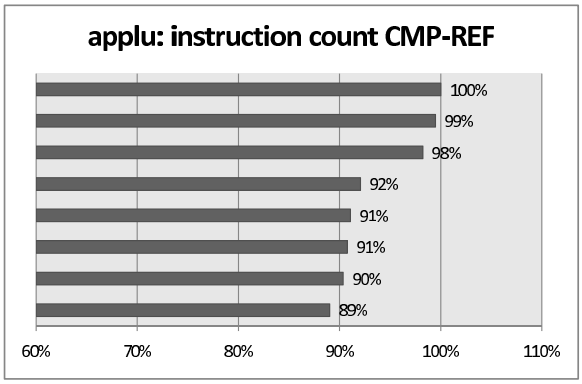
Fig. 5.15. applu: Instruction Count Distribution Across Cores



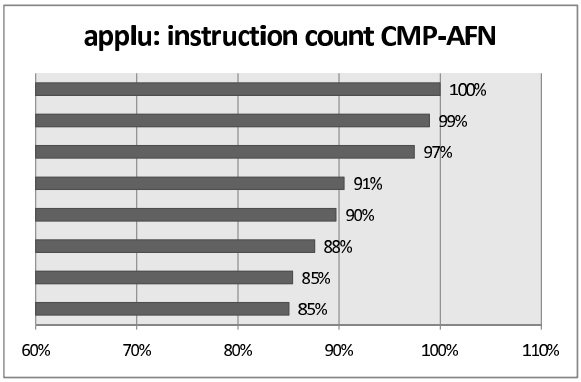
(a) 4-core CMP-REF



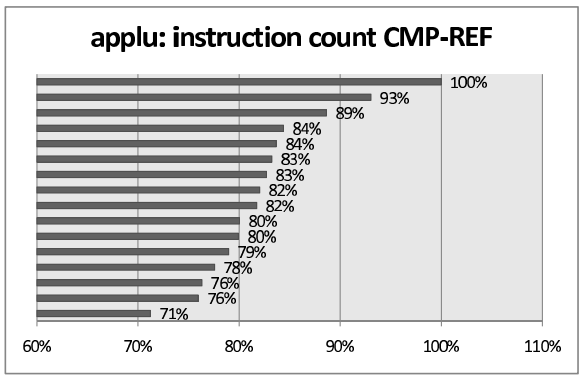
(b) 4-core CMP-AFN



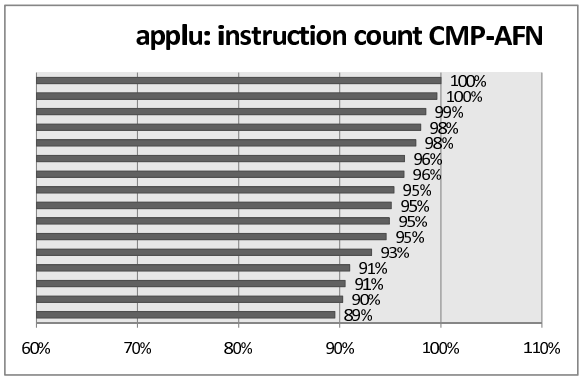
(c) 8-core CMP-REF



(d) 8-core CMP-AFN



(e) 16-core CMP-REF



(f) 16-core CMP-AFN

Fig. 5.16. applu: Instruction Count Distribution Across Cores (Normalized to Max Instruction Count)

6. SUMMARY

6.1 Conclusions

We have described the CMP-AFN architecture and the corresponding instruction set architecture (ISA) extensions that augment a shared memory chip multiprocessor with an aggregate function network and an interface between CPU cores and the on-chip AFN. In the CMP-AFN architecture, collective communications are performed without using or interfering with the on-chip cache coherent shared memory hierarchy. We have shown the on-chip synchronous aggregate communication model, implemented using the on-chip aggregate network hardware and ISA extensions, provide the CMP with low latency collective operations and reduced network traffic, resulting in effective use of on-chip cache and low power consumption.

Although this dissertation does not focus on gate-level architectural implementation issues, the abstract architecture, instruction set, and programmer model for the CMP-AFN design presented here represents a major advance. Efficiently managing the complexity of virtualizing processors to processes, and allowing them to dynamically move between cores (or cluster nodes) due to operating system scheduling have been open problems in AFN design for over a decade. Further, the instruction set extensions developed in this dissertation are shown to be straightforward and effective to use, and were tested as a fully integrated component of a GCC/GOMP OpenMP compilation and Linux runtime environment.

The detailed simulation results for the EPCC OpenMP microbenchmarks and SPEC OpenMP benchmarks both clearly and directly show a dramatic improvement in execution time and cache activity. The microbenchmarks revealed an order of magnitude improvement in all costs associated with barrier synchronization, better than 2X improvement in locks, and a relatively minor improvement in reductions. The

SPEC benchmark speedups ranged from negligible to better than 2X, with greater improvement over the reference CMP design as the number of cores increased from 4 to 16. We believe that it would continue to improve as more cores are added, but our simulation environment could not handle more cores. The simulation environment cannot quantify nor directly support our claim of reduced power consumption, however, the measured reductions in cache and memory activity are well known to imply reductions in power consumption.

In summary, if shared-memory CMPs are to scale to many cores, the concept of a CMP-AFN has been shown to be both a viable and effective means towards achieving this goal.

6.2 Future Work

Future work includes the performance evaluation of many-core (17-core or more) with the on-chip AFN and the exploration of off-chip AFN to connect multiple CMP-AFN sockets.

LIST OF REFERENCES

LIST OF REFERENCES

[1] G. F. Pfister and V. A. Norton, ""Hot Spot" Contention and Combining in Multistage Interconnection Networks," in *ICPP*, pp. 790–797, 1985.

[2] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[3] U. Drepper, "Futexes Are Tricky," Jan 2008. http://people.redhat.com/ drepper/futex.pdf.

[4] H. Franke, R. Russell, and M. Kirkwood, "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux," in *Proceedings of the 2002 Ottawa Linux Summit*, 2002.

[5] D. Novillo, "OpenMP and automatic parallelization in GCC," Jun 2006. GCC Developers' Summit, Ottawa, Ontario CANADA.

[6] V. Aslot and R. Eigenmann, "Performance Characteristics of the SPEC OMP2001 Benchmarks," *SIGARCH Comput. Archit. News*, vol. 29, no. 5, pp. 31–40, 2001.

[7] J. M. Bull and D. O'Neill, "A Microbenchmark Suite for OpenMP 2.0," *SIGARCH Comput. Archit. News*, vol. 29, no. 5, pp. 41–48, 2001.

[8] K. Fürlinger, M. Gerndt, and J. Dongarra, "Scalability Analysis of the SPEC OpenMP Benchmarks on Large-Scale Shared Memory Multiprocessors," in *Proceedings of the 2007 International Conference on Computational Science (ICCS 2007)*, (Beijing, China), pp. 815–822, May 2007.

[9] J. Sampson, R. Gonzalez, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 235–246, IEEE Computer Society, 2006.

[10] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[11] H. G. Dietz, T. M. Chung, and T. I. Mattox, "A Parallel Processing Support Library Based on Synchronized Aggregate Communication," in *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, (London, UK), pp. 254–268, Springer-Verlag, 1996.

[12] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer Designing a MIMD Shared Memory Parallel Computer," *IEEE Trans. on Computers*, vol. C-32, pp. 175–189, Feb 1983.

[13] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. S. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," in *ICPP*, pp. 764–771, 1985.

[14] R. E. Kessler and J. L. Schwarzmeier, "Cray T3D: A New Dimension for Cray Research," in *the 38th IEEE Computer Society International Conference (COMPCON)*, pp. 176–182, 1993.

[15] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak, "The Network Architecture of the Connection Machine CM-5 (extended abstract)," in *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, (New York, NY, USA), pp. 272–285, ACM Press, 1992.

[16] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "CEDAR: A Large Scale Multiprocessor," *SIGARCH Comput. Archit. News*, vol. 11, no. 1, pp. 7–11, 1983.

[17] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, "Overview of the Blue Gene/L system architecture," *IBM Journal of Research and Development*, vol. 49, no. 2-3, pp. 195–212, 2005.

[18] R. Hoare, H. G. Dietz, T. I. Mattox, and S. P. Kim, "Bitwise Aggregate Networks," in *SPDP '96: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, (Washington, DC, USA), p. 306, IEEE Computer Society, 1996.

[19] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures," in *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), pp. 341–353, IEEE Computer Society, 2003.

[20] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," in *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, (New York, NY, USA), pp. 422–433, ACM, 2003.

[21] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 291–302, IEEE Computer Society, 2003.

[22] G. S. Tyson, M. K. Farrens, and A. R. Pleszkun, "MISC: A Multiple Instruction Stream Computer," in *MICRO 25: Proceedings of the 25th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 193–196, 1992.

[23] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen, "Multiple Instruction Stream Processor," in *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pp. 114–127, 2006.

[24] J. Held, J. Bautista, and S. Koehl, "From a Few Cores to Many: A Tera-Scale Computing Research Overview," 2006. White Paper, Intel Corporation (http://www.intel.com/research/platform/terascale).

[25] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the CELL Multiprocessor," *IBM Journal of Research Development*, vol. 49, no. 4/5, pp. 589–604, 2005.

[26] M. K. Velamati, A. Kumar, N. Jayam, G. Senthilkumar, P. K. Baruah, R. Sharma, S. Kapoor, and A. Srinivasan, "Optimization of Collective Communication in Intra-Cell MPI," in *HiPC*, pp. 488–499, 2007.

[27] R. Nishtala and K. A. Yelick, "Optimizing Collective Communication on Multicores," *First USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, 2009.

[28] T. Johnson and U. Nawathe, "An 8-core, 64-thread, 64-bit power efficient sparc soc (Niagara2)," in *ISPD '07: Proceedings of the 2007 international symposium on Physical design*, (New York, NY, USA), pp. 2–2, ACM, 2007.

[29] M. J. Quinn, *Parallel Computing (2nd ed.): Theory and Practice*. New York, NY, USA: McGraw-Hill, Inc., 1994.

[30] H. Dietz, *Linux Parallel Processing HOWTO*. Bloomington, IN, USA: iUniverse, Inc., 2000. (http://aggregate.org/LDP/19980105/pphowto.html).

[31] OpenMP Architecture Review Board. http://www.openmp.org.

[32] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, 1998.

[33] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 2.5*, May 2005. (http://www.openmp.org/mp-documents/spec25.pdf).

[34] J. P. Hoeflinger, "Extending OpenMP to Clusters," 2006. White paper, Intel Corporation.

[35] S.-J. Min, A. Basumallik, and R. Eigenmann, "Optimizing OpenMP programs on software distributed shared memory systems," *Int. J. Parallel Program.*, vol. 31, no. 3, pp. 225–249, 2003.

[36] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, 1996.

[37] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI The Complete Reference: Volume 2, The MPI-2 Extensions*. Cambridge, MA: MIT Press, 1998.

[38] H. G. Dietz, T. I. Mattox, and G. Krishnamurthy, "The Aggregate Function API: It's Not Just for PAPERS Anymore.," in *LCPC*, pp. 277–291, 1997.

[39] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. Santa Clara, CA, order number: 253665-023us ed., May 2007.

[40] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.

[41] S. L. Scott, "Synchronization and Communication in the T3E Multiprocessor," in *ASPLOS-IX: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–36, 1996.

[42] N. E. Jerger, L.-S. Peh, and M. Lipasti, "Virtual Circuit Tree Multicasting: A Case for On-Chip Hardware Multicast Support," in *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 229–240, IEEE Computer Society, 2008.

[43] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[44] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[45] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," in *WOMPAT '01: Proceedings of the International Workshop on OpenMP Applications and Tools*, (London, UK), pp. 1–10, Springer-Verlag, 2001.

[46] H. Saito, G. Gaertner, W. Jones, R. Eigenmann, H. Iwashita, R. Lieberman, M. van Waveren, and B. Whitney, "Large system performance of SPEC OMP benchmark suites," *Int. J. Parallel Program.*, vol. 31, no. 3, pp. 197–209, 2003.

[47] Standard Performance Evaluation Corporation, "SPEC OMP (OpenMP Benchmark Suite)," 2001. (http://www.spec.org/hpg/omp2001).

[48] GOMP (GNU OpenMP) – An OpenMP Implementation for GCC. http://gcc.gnu.org/projects/gomp.

[49] OpenMP Architecture Review Board, *OpenMP Application Program Interface, Version 3.0*, May 2008. Available from www.openmp.org.

[50] EPCC, The University of Edinburgh, "EPCC OpenMP Microbenchmarks," 1999. (http://www.epcc.ed.ac.uk/research/openmpbench).

[51] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M.* Santa Clara, CA, order number: 253666-023us ed., May 2007.

[52] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z.* Santa Clara, CA, order number: 253667-023us ed., May 2007.

APPENDICES

A. AFN EXTENSIONS TO IA-32

A.1 Overview of ISA Extensions for on-chip AFN

AFN extensions use the aggregate function (AF) communication model [38]. The extensions support inter-processor data communication, barrier synchronization, and synchronous collective operations via the on-chip Aggregate Function Network (AFN).

If CPUID.01H:ECX.AFN[bit TBD] = 1, AFN extensions are present. (bits 16-31 are available as of Fall 2007 [51, 52]).

AFN extensions add the following features to the IA-32 architecture, while maintaining backward compatibility with all existing IA-32 processors, applications and operating systems.

- Instructions to support the following synchronous collective operations (i.e. aggregate functions) among a team of threads:
  - Barrier synchronization instruction
  - Multi-broadcast data transfer instructions among a team of threads (16-bit, 32-bit integer data and floating-point data)
  - Reduction instructions (addition and production on integer and floating point data)
  - Instructions that save and restore the state of the AFNCSR and AFNSAR registers
- Modifications to existing IA-32 instructions to support AFN features:
  - Extensions and modifications to the CPUID instruction
  - (Optional) Modifications to the RDPMC instruction

These new features extend the IA-32 architectures MP (multiprocessor) programming in the following important ways:

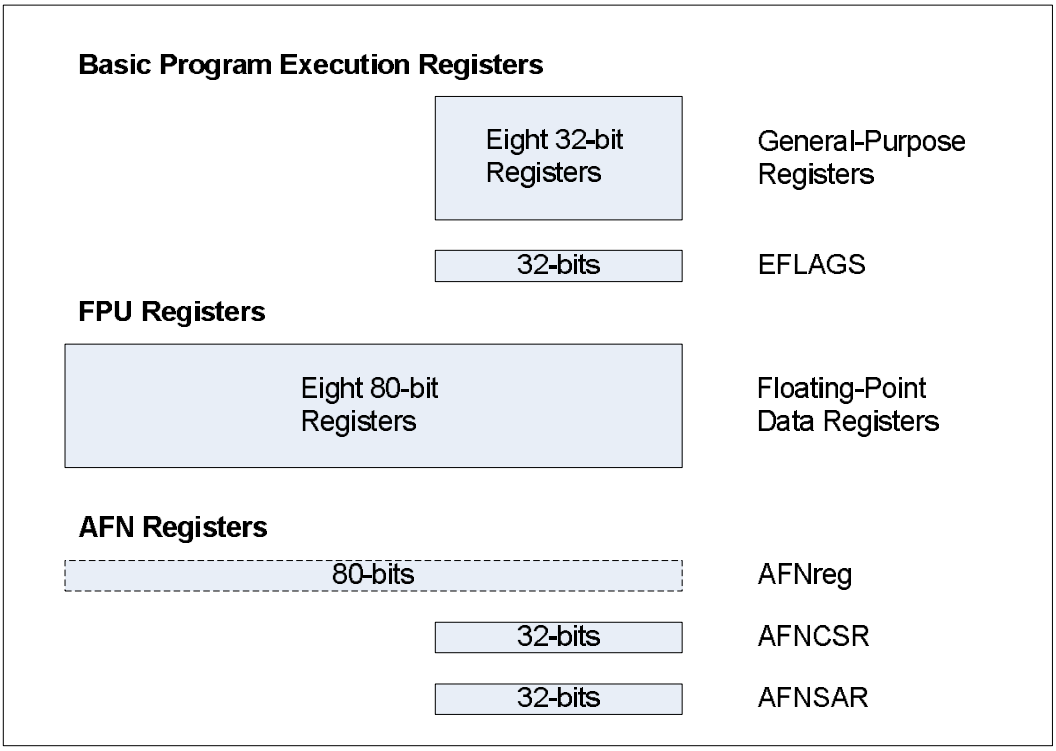


Fig. A.1. AFN Programming Environment: a single thread's perspective

- They provide the ability to perform integer (16- or 32-bit) or floating-point data transfer among a team of threads, without requiring control and data traffic in cache memory hierarchy.
- They provide the ability to perform synchronous collective operations without requiring control and data traffic in cache memory hierarchy.

The AFN extensions are fully compatible with all software written for IA-32 processors. All existing software continues to run correctly, without modification, on processors that incorporate AFN extensions, as well as in the presence of applications that incorporate these extensions.

A.2 AFN Programming Environment

Figure A.1 shows the programming environment of the AFN extensions. All AFN instructions use general-purpose registers, floating-point registers, or AFNreg for both source and destination registers. There is only one AFNreg that can be read from, or written to, by the processor.

The programming environment of the AFN extension consists of the followings:

- Eight 32-bit general purpose registers.
- Eight 80-bit floating point data registers.
- An 80-bit AFNreg (added for the AFN extension) - One of AFNreg's that is assigned to the processor. Each processor can read and write from/to only one AFNreg. The processor moves data to the AFNreg for the AFN operations, then reads the result from the AFNreg when the result is ready. The AFN performs aggregate function operations on contents of all AFNreg's and places the result in the AFNreg.
- A 32-bit AFNSAR register (added for the AFN extension) - stores an 8-bit afuID, an 8-bit rank, and a 16-bit secureID. The register can be written in privilege mode. All AFN instructions read AFNSAR and send it to the AFN.
  - the 8-bit afuID (stored in AFNSAR register): indicates the hardware (AFU) that is assigned to a team of threads for aggregate (collective) functions. As shown in Figure A.2, an AFN (Aggregate Function Network) consists of multiple AFUs (Aggregate Function Units) to handle multiple teams of threads at the same time.
  - the 8-bit rank (stored in AFNSAR register): indicates the unique ID that is assigned to the thread within a team of threads.
  - the 16-bit secureID (stored in AFNSAR register): used for the AFN extensions security features.

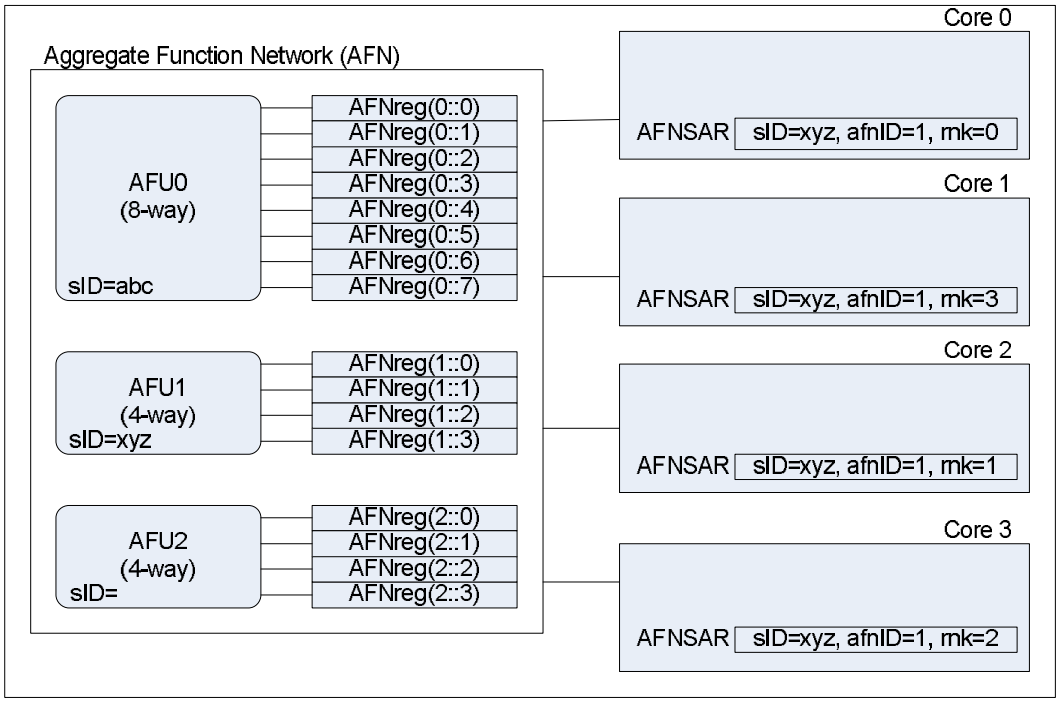


Fig. A.2. AFNregs and AFUs

- A 32-bit AFNCSR register (added for the AFN extension) - the 32-bit register (see Figure A.3) provides status and control bits used in AFN floating-point operations.

A.2.1 AFU

An AFN implementation may have one or more AFUs (Aggregate Function Units). Each AFU can process a team of threads independently and an AFN with multiple AFUs can handle multiple teams of threads simultaneously. Figure A.2 shows an AFN implementation that has three AFUs.

A.2.2 AFNreg Registers

There are a number of AFNreg registers in each AFU. The number of AFNreg registers in an AFN determines the maximum size of the team of threads. Each AFNreg register in an AFU is referenced by rank within the afuID. For example, AFNreg(x::y) indicates the AFNreg register for AFUID=x and rank=y. afuID indicates one of the AFUs, where AFN operations are performed for the processor, and rank indicates the unique integer that is assigned to the processor within a team of threads.

Each processor can read from and write to only one AFNreg register and cannot access other AFNreg registers. AFNreg refers to the AFNreg register that is accessible by the processor within the processor context.

The 8-bit afuID and the 8-bit rank are stored in AFNSAR register bits [15:0]. Instructions that read and write from/to the AFNreg do not have AFNreg as an operand.

A.2.3 AFNSAR Register

The AFNSAR register stores the 16-bit *s*ecureID, the 8-bit *a*fuID, and the 8-bit *r*ank. The contents of AFNSAR register can only be written by the AFNALLOC instruction, which is executed in privilege mode only. The bits of the AFNSAR are:

- secureID (bits 31-16):
- afuID (bits 15-8):
- rank (bits 7-0):

secureID is not an explicit operand, but the CPU core that executes the AFN instructions reads the secureID from the AFNSAR and sends it to the AFN for authentication. This is a security feature to prevent malicious threads from manipulating the secureID.

The process ID (pid) of the process that owns the team of threads can be used for the secureID value. secureID is also stored in the AFN.

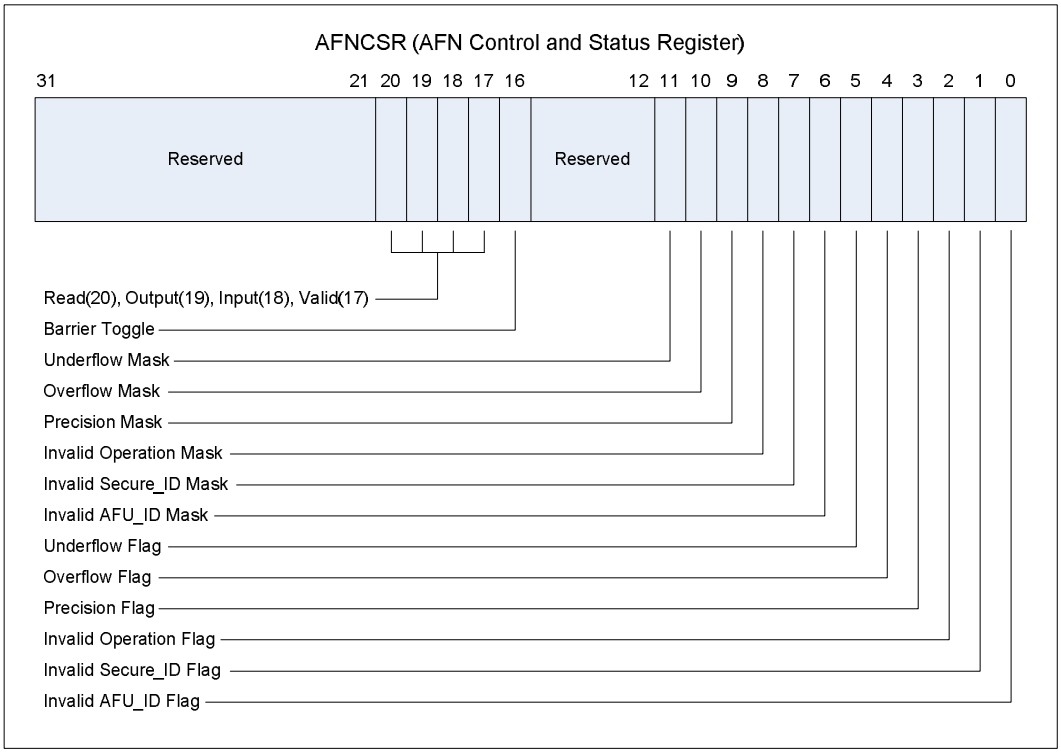


Fig. A.3. AFNCSR Control/Status Register

A.2.4 AFNCSR Control and Status Register

The 32-bit AFNCSR register (see Figure A.3) contains control and status information for AFN operations. This register contains flag and mask bits for AFN operation exceptions.

The contents of this register can be loaded from memory with the LDAFNCSR instruction and stored in memory with STAFNCSR.

Bits 12 through 15 and 21 through 31 of the AFNCSR register are reserved and are cleared on a power-up or reset of the processor.

Bits 0 through 5 of the AFNCSR register indicate whether an AFN operation exception has been detected. They are sticky flags. That is, after a flag is set, it remains set until explicitly cleared. To clear these flags, use the LDAFNCSR instruction to write zeros to them.

Bits 6 through 11 provide individual mask bits for the AFN operation exceptions. An exception type is masked if the corresponding mask bit is set, and it is unmasked if the bit is clear. These mask bits are set upon a power-up or reset. This causes all AFN operation exceptions to be initially masked.

Bit 16 represents the barrier number that is toggled.

Bits 17 through 20 represent the status of the AFN instruction for the AFNreg – *valid* (bit 17), *reset* or *wait for input* (bit 18), *input arrived* (bit 19), and *output ready* (bit 20).

```
--------+-----------------------------------------
2 1 1 1
0 9 8 7

o i r v
u n e a
t p s l
p u e i
u t t d
t
--------+-----------------------------------------
0 0 0 0   Not participate in barrier group
0 0 1 1   Waiting for input (afnOP) to arrive
0 1 0 1   Input (afnOP) arrived, output is not ready
1 0 0 1   Output is ready for afnRD
--------+-----------------------------------------
```

Fig. A.4. AFNCSR register bit positions

A.3 AFN Instructions

AFN instructions are divided into five functional groups (Opcode range):

- Data Movement Instructions (0F 38 8x):
  - AFNRD – Read AFNreg
  - AFNPUTGET – Multi-broadcast data communication
- Synchronization instruction (0F 38 9x):
  - AFNBARR – Barrier Synchronization
  - AFNBARRRD – Barrier Synchronization Read
  - AFNLOCK – Mutex lock
  - AFNUNLOCK – Mutex unlock
- Reduction instructions (0F 38 Ax,Bx,Cx,Dx):
  - AFNADD/AFNFADD/AFNFADDP – Reduce ADD
  - AFNMUL/AFNFMUL/AFNFMULP – Reduce Multiply
  - AFNAND - Reduce bitwise AND
  - AFNOR - Reduce bitwise OR
  - AFNXOR - Reduce bitwise XOR
- State management instructions (0F 38 Ex):
  - LDAFNCSR - Load AFNCSR Register
  - STAFNCSR - Store AFNCSR Register State
  - LDAFNSAR - Load AFNSAR Register
  - STAFNSAR - Store AFNSAR Register State
- AFN configuration instructions (0F 38 Fx):
  - AFNALLOC - Allocate AFU
  - AFNFREE - Free AFU

A.3.1 Opcodes

In IA-32 architecture, a primary opcode can be 1, 2, or 3 bytes in length. Three-byte opcode formats for general-purpose and SIMD instructions consist of:

- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes, or
- A mandatory prefix (66H), an escape opcode byte, plus two additional opcode bytes (same as previous bullet)

For example, PHADDW for MMX registers consists of the following sequence: 0F 38 01.

As of August 2007 [51, 52], the following three-byte opcodes are not being used and the opcodes for AFN instructions were assigned to 0F 38 80..FF:

- 0F 38 00..0B
- 0F 38 1C..1E
- 0F 3A 0F

A.3.2 Instruction Format

The following is an example of the format used for each instruction description in this chapter. The heading below introduces the example. The table below provides an example summary table.

AFNBARR – Barrier Synchronization [this is an example]

A.3.3 Opcode Column in the Instruction Summary Table

The Opcode column in the instruction summary table (shown in Figure A.5) shows the object code produced for each form of the instruction. When possible, codes are given as hexadecimal bytes in the same order that they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

Opcode	Instruction	Description
0F 38 90	AFNBARR EAX	Barrier synchronization

Fig. A.5. An Example Opcode Summary Table: consists of three columns – "Opcode", "Instruction", and "Description"

- /r - Indicates that the ModR/M byte[1] of the instruction contains a register operand and an r/m operand.
- ib, iw, id, io - A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words, doublewords and quadwords are given with the low-order byte first.
- +i - A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

A.3.4 Instruction Column in the Instruction Summary Table

The Instruction column in the instruction summary table (shown in Figure A.5) gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

- AFNreg - The AFN register that is assigned to the processor. AFNreg == AFNreg(AFNSAR[15:8]::AFNSAR[7:0]).
- AFNreg(x::y) - The y^{th} AFN register in AFU x.
- AFNreg(::y) - indicates AFNreg(x::y), where x == AFNSAR[15:8].
- r8 - One of the byte general-purpose registers: AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPA, DILL and SIL; or one of the byte registers (R8L - R15L) available when using REX.R and 64-bit mode.
- r16 - One of the word general-purpose registers: AX, CX, DX, BX, SP, BP, SI, DI; or one of the word registers (R8-R15) available when using REX.R and 64-bit mode.

[1] the ModR/M byte refers to an addressing-form specifier byte following the primary opcode.

- r32 - One of the doubleword general-purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESC, EDT; or one of the doubleword registers (R8D - R15D) available when using REX.R in 64-bit mode.
- r64 - One of the headword general-purpose registers: RAD, RX, RCS, RD, RID, REI, RAP, RASP, R8R15. These are available when using REX.R and 64-bit mode.
- imm8 - An immediate byte value. The imm8 symbol is a signed number between 128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.
- imm16 - An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between 32,768 and +32,767 inclusive.
- imm32 - An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and 2,147,483,648 inclusive.
- imm64 - An immediate headword value used for instructions whose operand-size attribute is 64 bits. The value allows the use of a number between +9,223,372,036,854,775,807 and 9,223,372,036,854,775,808 inclusive.
- r/m8 - A byte operand that is either the contents of a byte general-purpose register (AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPA, DILL and SIL) or a byte from memory. Byte registers R8L - R15L are available using REX.R in 64-bit mode.
- r/m16 - A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, CX, DX, BX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation. Word registers R8W - R15W are available using REX.R in 64-bit mode.

- m16 - A word operand in memory, usually expressed as a variable or array name, but pointed to by the DES:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- m32 - A doubleword operand in memory, usually expressed as a variable or array name, but pointed to by the DES:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- m32fp, m64fp, m80fp - A single-precision, double-precision, and double extended-precision (respectively) floating-point operand in memory. These symbols designate floating-point values that are used as operands for x87 FU floating-point instructions.
- ST or ST(0) - The top element of the FU register stack.
- ST(i) - The i^{th} element from the top of the FU register stack ($i \leftarrow 0$ through 7).
- XML - An XML register. The 128-bit XML registers are: XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode.

AFNRD – Read AFNreg

Opcode	Instruction	Description
0F 38 8A	AFNRD r32	Move AFNreg to r32
0F 38 80+i	AFNRD ST(i)	Move AFNreg to ST(i)
0F 38 88	AFNRD ST(0)	Push AFNreg onto the FU register stack

Description

Copies the *AFNreg* to the operand (destination operand) if the content of AFNreg is ready to be read (i.e. AFNCSR[19:17]=0b111). The destination register can be a general purpose register or the top of the FU register stack. The operand can be a word, a doubleword or an 80-bit floating point (for double extended-precision floating point). If the AFNreg is not ready (i.e. AFNCSR[19:17]=0b011), no data is moved to the destination operand.

Operation

```
IF AFNreg is ready
    THEN
    IF DEST is ST(i)
        THEN
            TOP <-- TOP - 1;
            ST(0) <-- AFNreg;
        ELSE
            DEST <-- AFNreg;
    FI;
FI;
```

AFNPUTGET – Multi-broadcast Data Transfer

```
Opcode      Instruction          Description
0F 38 88 /r AFNPUTGET r32,r32    Move AFNreg(SRC2) to AFNreg
0F 38 8A /r AFNPUTGET ST(i),r32  Move AFNreg(SRC2) to AFNreg
```

Description

Performs data transfer among AFNreg(::i). AFNPUTGET moves the first operand (source operand 1) to AFNreg, then moves AFNreg(SRC2) into AFNreg when all AFNreg(::i)'s have arrived at the AFN. The second operand (SRC2) represents the rank of the processor where the data comes from. AFNPUTGET should be followed by AFNRD instruction to complete a multi-broadcast PutGet operation.

The *reg/opcode* field of ModR/M byte specifies the second operand. The *r/m* field of ModR/M byte specifies a register for the first operand. For example, for "AFNPUTGET ST(i),r32" instruction, the 3-bit R/M field of ModR/M byte indicates one of the FP data registers.

Operation

```
AFNreg <-- SRC1;
Barrier Sync;
temp <-- AFNreg(::SRC2);
AFNreg <-- temp;
```

AFNBARR – Barrier Synchronization

Opcode	Instruction	Description
0F 38 90	AFNBARR EAX	Barrier synchronization
		EDX holds afuID and rank
0F 38 91	AFNBARRRD EAX	Barrier Synchronization

Description

Performs barrier synchronization via AFN. There is no operand for this instruction. `AFNBARR` should be followed by the `AFNBARRRD` instruction to complete the barrier synchronization operation.

Operation

```
Barrier Sync;
AFNreg <-- TRUE;
```

AFNLOCK – Mutex Lock

Opcode	Instruction	Description
0F 38 98	AFNLOCK	AFN mutex lock
		EDX holds afuID and rank

Description

Performs a mutex lock via AFN. If the lock is successful, it stores a non-zero value to EAX. Otherwise, it stores zero to EAX.

Operation

```
IF (currState==LOCKED)
    THEN
        // Unsuccessful
        EAX <- zero
FI
IF (currState==UNLOCKED)
    THEN
        // Successful
        EAX <- non-zero
        currState <- LOCKED
        lockOwner <- rank
FI
```

AFNUNLOCK – Mutex Unlock

Opcode	Instruction	Description
0F 38 99	AFNUNLOCK	AFN Mutex Unlock
		EDX holds afuID and rank

Description

Performs a mutex unlock via AFN. If the unlock is successful, it stores non-zero value to EAX. Otherwise, it stores zero to EAX. The unlock is successful if current status is LOCKED and lockOwner is equal to the rank.

Operation

```
IF (currState==LOCKED and lockOwner==rank)
    THEN
        // Successful
        EAX <- non-zero
    ELSE
        // Unsuccessful
        EAX <- zero
FI
```

AFNADD/AFNFADD/AFNFADDP – Reduce ADD

Opcode	Instruction	Description
0F 38 AA	AFNADD r32	Move r32 to AFNreg for reduction ADD
0F 38 A0+i	AFNFADD ST(i)	Move ST(i) to AFNreg for reduction ADD
0F 38 A8	AFNFADDP	Move ST(0) to AFNreg for reduction ADD, and pop the register stack

Description

Performs a reduction add operation. `AFNADD/AFNFADD/AFNFADDP` copies the operand (source operand) to the *AFNreg*. The AFN adds all AFNreg's from all threads. The source register can be a general purpose register or floating-point register stack. The operand can be a word, a doubleword, or an 80-bit floating point number (for double extended-precision floating point).

The no-operand version of the instruction sends the content of ST(0) to the AFN for the reduction add and performs the additional operation of popping the FU register stack after moving the top of stack content to AFNreg.

Operation

```
AFNreg <-- SRC;
IF Instruction = AFNFADDP
    THEN
        PopRegisterStack;
FI;
Barrier Sync;
temp <-- (AFNreg(::0) + AFNreg(::1) + ... + AFNreg(::Nthreads-1));
AFNreg <-- temp;
```

AFNMUL/AFNFMUL/AFNFMULP – Reduce Multiply

Opcode	Instruction	Description
0F 38 BA	AFNMUL r32	Move r32 to AFNreg for reduction MUL
0F 38 B0+i	AFNFMUL ST(i)	Move ST(i) to AFNreg for reduction MUL
0F 38 B8	AFNFMULP	Move ST(0) to AFNreg for reduction MUL, and pop the register stack

Description

Performs a reduction multiplication operation. `AFNMUL/AFNFMUL/AFNFMULP` copies the operand (source operand) to the *AFNreg*. The AFN multiplies all AFNreg's from all threads. The source register can be a general purpose register or floating-point register. The operand can be a word, a doubleword, or an 80-bit floating point data (for double extended-precision floating point).

The no-operand version of the instruction sends the content of ST(0) to the AFN for the reduction multiplication and performs the additional operation of popping the FU register stack after sending the top of stack content to AFNreg.

Operation

```
AFNreg <-- SRC;
IF Instruction = AFNFMULP
    THEN
        PopRegisterStack;
FI;
Barrier Sync;
temp <-- (AFNreg(::0) x AFNreg(::1) x ... x AFNreg(::Nthreads-1));
AFNreg <-- temp;
```

AFNAND – Reduce bitwise AND

Opcode	Instruction	Description
0F 38 C0	AFNAND r32	Move r32 to AFNreg for reduction bitwise AND

Description

Performs a reduction bitwise AND. `AFNAND` copies the operand (source operand) to the *AFNreg*. The AFN performs bitwise logical AND on all AFNreg's from all threads. The source register can be a general purpose register. The operand can be a word, or a doubleword.

Operation

```
AFNreg <-- SRC;
Barrier Sync;
temp <-- (AFNreg(::0) AND AFNreg(::1) AND ... AND AFNreg(::Nthreads-1));
AFNreg <-- temp;
```

AFNOR – Reduce bitwise OR

Opcode	Instruction	Description
0F 38 C4	AFNOR r32	Move r32 to AFNreg for reduction bitwise OR

Description

Performs a reduction bitwise OR. `AFNOR` copies the operand (source operand) to the *AFNreg*. The AFN performs bitwise logical OR on all AFNreg's from all threads. The source register can be a general purpose register. The operand can be a word, or a doubleword.

Operation

```
AFNreg <-- SRC;
Barrier Sync;
temp <-- (AFNreg(::0) OR AFNreg(::1) OR ... OR AFNreg(::Nthreads-1));
AFNreg <-- temp;
```

AFNXOR – Reduce bitwise XOR

Opcode	Instruction	Description
0F 38 C8	AFNXOR r32	Move r32 to AFNreg for reduction bitwise XOR

Description

Performs a reduction bitwise XOR. `AFNXOR` copies the operand (source operand) to the *AFNreg*. The AFN performs bitwise logical XOR on all AFNreg's from all threads. The source register can be a general purpose register. The operand can be a word, or a doubleword.

Operation

```
AFNreg <-- SRC;
Barrier Sync;
temp <-- (AFNreg(::0) XOR AFNreg(::1) XOR ... XOR AFNreg(::Nthreads-1));
AFNreg <-- temp;
```

LDAFNCSR – Load AFNCSR Register

Opcode	Instruction	Description
0F 38 E0	LDAFNSAR m32	Load AFNSAR register from m32.

Description

AFNCSR loads the source operand into the AFNCSR control and status register. The source operand is a 32-bit memory location. See refTBD for a description of the AFNCSR register and its contents. The LDAFNCSR instruction is typically used in conjunction with the STAFNCSR instruction, which stores the contents of the AFNCSR register in memory. The default AFNCSR value at reset is TBD.

Operation

```
AFNCSR <-- m32;
```

STAFNCSR – Store AFNCSR Register State

Opcode	Instruction	Description
0F 38 E1	STAFNCSR m32	Store the content of AFNCSR register to m32.

Description

Stores the contents of the AFNCSR control and status register in the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the AFNCSR register are stored as 0s.

Operation

```
m32 <-- AFNCSR;
```

LDAFNSAR – Load AFNSAR Register

Opcode	Instruction	Description
0F 38 E2	LDAFNSAR m32	Load AFNSAR register from m32.

Description

AFNSAR loads the source operand into the AFNSAR register. The source operand is a 32-bit memory location. See refTBD for a description of the AFNSAR register and its contents. The LDAFNSAR instruction is typically used in conjunction with the STAFNSAR instruction, which stores the contents of the AFNSAR register in memory. The default AFNCSR value at reset is TBD.

Operation

```
AFNSAR <-- m32;
```

STAFNSAR – Store AFNSAR Register State

Opcode	Instruction	Description
0F 38 E3	STAFNSAR m32	Store the content of AFNSAR register to m32.

Description

Stores the contents of the AFNSAR register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the AFNSAR register are stored as 0s.

Operation

```
m32 <-- AFNSAR;
```

AFUALLOC – Allocate an AFU

Opcode	Instruction	Description
0F 38 F0	AFUALLOC r32	Allocate an AFU.

Description

Allocates an AFU for a team of threads. The first operand (source operand) represents the number of processors/threads for the team. AFUALLOC can only be executed in privilege mode. AFUALLOC sends secureID to AFU for the security feature and AFU stores the secureID in its local storage.

When a new team of threads is created, an AFU is assigned via AFUALLOC instruction. The AFN returns the afuID to the CPU core that executes the AFUALLOC instruction. When no AFU is available for the new team, the AFN clears the most significant bit of the afuID.

The user AFN operations will trap if the most significant bit of the afuID is clear – msb(afuID)=0–, then AFN operations will be performed via software, not via AFN hardware.

Operation

```
Send nThreads (SRC) to AFN
Receive afuID from AFN
IF allocation successful // i.e. msb(afuID)==TRUE
    THEN
        Write afuID to AFNSAR
        Send secureID to AFU
        Initialize AFU
FI;
```

AFUFREE – Free up an AFU

Opcode	Instruction	Description
0F 38 F1	AFUFREE	Free up an AFU.

Description

Frees up the AFU. AFUFREE does not have operands. The CPU that executes AFUFREE instruction extracts the afuID from AFNSAR and then sends the afuID to the AFN.

Operation

...

VITA

VITA

To be added...