# A HIGH-LEVEL PROGRAMMING MODEL FOR EMBEDDED MULTICORE PROCESSORS

---

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

---

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

By

Ayodunni Aribuki

August 2013

# A HIGH-LEVEL PROGRAMMING MODEL FOR EMBEDDED MULTICORE PROCESSORS

Ayodunni Aribuki

APPROVED:

Dr. Ernst L. Leiss, Chairman
Dept. of Computer Science

Dr. Edgar Gabriel
Dept. of Computer Science

Dr. Lennart Johnsson
Dept. of Computer Science

Dr. Jehan-François Pâris
Dept. of Computer Science

Dr. Eric Stotzer
Texas Instruments

Dean, College of Natural Sciences and Mathematics

# A HIGH-LEVEL PROGRAMMING MODEL FOR EMBEDDED MULTICORE PROCESSORS

---

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

---

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

By

Ayodunni Aribuki

August 2013

# Abstract

Traditionally, embedded programmers have relied on using low-level mechanisms for coordinating parallelism and managing memory. This is typically a herculean task, especially considering that this approach is processor-specific and requires that the process must be redone to target different deployment processors. As multicore technology becomes more prevalent in embedded systems, high-level approaches are being sought to reduce programmers' burden as they write code for more complex multicore systems. This dissertation explores implementing a high-level shared-memory parallel programming model for embedded multicore processors. The processor representative of this type that is used for this work is the TMS320C6678 (also referred to as C6678) digital signal processor (DSP) manufactured by Texas Instruments.

The C6678 is a high-performance fixed and floating-point DSP that comprises eight DSP core subsystems. In addition to external memory, it has roughly 8MB of on-chip memory, most of which may be configured as either cache or scratch-pad. When a portion of its local on-chip memory is configured as cache, software-controlled mechanisms must be used to manage the coherence of shared data that is cached in core-local memories. When the same memory is configured as scratch-pad, software-controlled mechanisms are also necessary to manage data movements between memory segments within the memory hierarchy. This memory organization brings additional challenges when developing applications for the C6678 as well as other processors with similar memory setups.

In this dissertation, we present a compiler implementation of a high-level programming model for managing parallelism in the C6678. This implementation is

leveraged to automatically utilize scratchpad memory without additional intervention from the programmer. A high-level construct is also introduced for controlling data placement. An assessment of the performance impact of various memory configurations of the C6678 is also presented.

# Acknowledgements

This dissertation would not exist without the contributions of many people. I would like to express my deep-felt thanks to my Ph.D. advisor, Dr. Ernst Leiss, for his thoughtful guidance and support throughout my graduate program, especially through the times when I was ready to quit. I thank Dr. Eric Stotzer for mentoring me at Texas Instruments and being willing to listen and discuss ideas with me. I also thank my other committee members — Drs. Paris, Johnsson, and Gabriel, for valuable feedback on my drafts and a diversity of perspectives. I am also indebted to Dr. Chapman for introducing me to research in programming languages and compilers.

I would also like to express sincere gratitude to current and past members of staff of the Computer Science Department especially Yvette Elder, Linda Robinson, Kim Jordan, Barbara Murray, Liz Faig, Anh Nash, and Jackie Baum. I am thankful to them for cheerfully assisting me with the administrative aspects that go along with getting a Ph.D. and providing moral support along the way.

I am blessed to be surrounded by loving family and friends. You know who you are, you helped me get here and I thank you. I especially thank Soji, who shared each step of my graduate experience with me. Finally, I thank my dear dad. His confidence in my abilities is unnerving, his advice is consistently timely, his love keeps me going.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Embedded Multicore

Multicore technology has been prominent in the high-performance, desktop, and gaming markets [33, 60, 34]. It is now finding its way into the embedded space. The major reason for this trend is the increasing demand for more powerful, but at the same time less power consuming processors to meet the ever increasing processing demands of embedded applications. Multicore is a shift from the traditional method of relying on higher speeds, which demanded higher power consumption to derive more performance. The Cortex-A9 MPCore from ARM [16], the MSC8156 Digital Signal Processor (DSP) from Freescale Semiconductors [29], and the TMS320C6678 DSP from Texas Instruments [72] are examples of embedded multicore processors that are currently used in medical, networking, telecommunications, and industrial computing applications.

While the multicore paradigm offers the obvious advantage of hardware parallelism, applications running on multicore processors will have to be carefully written to exploit this parallelism efficiently. Specifically, tasks such as decomposing the application into separate tasks that may be concurrently executed, mapping the execution of these tasks to cores, and managing communication and synchronization between tasks/cores must be performed.

Language designers have responded to the complexity of parallel programming by providing high-level languages/constructs that simplify and structure the way the programmer thinks about and expresses parallelism within programs. The use of high-level abstractions also shields the programmer from the low-level aspects associated with mapping computations to parallel machines. In most cases, this approach does not sacrifice performance when the platform implementation includes code optimization strategies. Examples of high-level shared memory parallel programming models include OpenMP [26], Intel's Threading Building Blocks (TBB) [67], and Cilk [21]. Of these, OpenMP is the de facto standard for parallel programming on shared memory systems and enjoys wide popularity (especially in the high performance computing community) because of its ease of use, support for incremental parallelism, performance portability, and wide availability.

While significant progress has been attained in boosting the performance of processors through parallel processing, high-latency off-chip memory accesses remains one of the biggest bottlenecks to the performance of applications in both single-core and multicore environments [65, 27]. Processor speed has been increasing at a more rapid rate than memory speed leading to the *memory wall* problem [77] where the

CPU stalls while waiting for memory requests to be fulfilled. In order to close the gap between processor and memory speed, general-purpose and embedded multicore processors incorporate fast on-chip static random access memory (SRAM), sometimes at multiple levels, as a means of reducing the number of slower off-chip (dynamic RAM) DRAM accesses. The idea is for the SRAM to hold a subset of an application's dataset/instructions for faster access. SRAM blocks may be operated as either caches or scratchpad memory. When the memory reference pattern of an application has good spatial and/or temporal reference locality (i.e., memory references are clustered in space and/or time), they can benefit from having on-chip SRAM.

General-purpose processors typically have multi-level hardware caches, usually with hardware cache coherence, where the hardware is responsible for moving data across levels of the memory hierarchy. However, caches are not always effective because they are subject to misses—there is no guarantee that a requested memory item is contained in the cache. A cache miss causes a stall in execution until the requested memory item is brought in from a secondary cache or main memory. Scratchpad memory is an alternative to caches that is common in embedded processors because it is not subject to misses, and therefore offers more deterministic performance. Scratchpad memories are also considered to be more energy and space efficient than caches [18]. Like caches, scratchpad memories are low-latency storage but rely on explicit software instructions to manage their content. Allocation and movement of memory objects in scratchpad must be managed either by the programmer or compiler.

Some of the processors incorporate both caches and scratchpad memories. High-end embedded processors often allow their memory organization to be flexibly configured such that a memory region may be configured entirely as cache, entirely as scratchpad, or as a combination of both. Examples of processors that have on-chip memory that may be configured either as cache or scratchpad include the P2020 and P2010 from Freescale Semiconductors [30] and the TMS320C6678 from Texas Instruments [72]. The Kepler GK110 from NVIDIA [59] also supports several scratchpad/-cache configurations. ARM's Cortex-A9 MPCore does not include scratchpad but allows cache sizes for each core to be independently configured [16]. This flexibility enables a processor's underlying memory system to be adapted to an application's requirements and provides opportunities for saving power [66, 78, 79].

## 1.2 TMS320C6678: A Representative Embedded Multicore Processor

The TMS320C6678 (subsequently, we will refer to this as C6678) DSP is a high-performance multicore fixed and floating-point DSP that is based on TI's Keystone multicore architecture [72]. It comprises eight C66x DSP core subsystems. Each of the cores runs at either 1 or 1.25GHz (depending on the version) and can perform 32 Multiply-Accumulates (MACs) per cycle for fixed-point computations and 16 single-precision floating-point operations per cycle when operating at 1.25GHz. The combination of the eight cores gives the C6678 a peak computational performance of 160 single-precision GFLOPS and 60 double-precision GFLOPS . With a

power consumption of just 10W, the C6678 can achieve up to 16 single-precision GFLOPS/Watt, making it one of the most power-efficient processors in the market today.

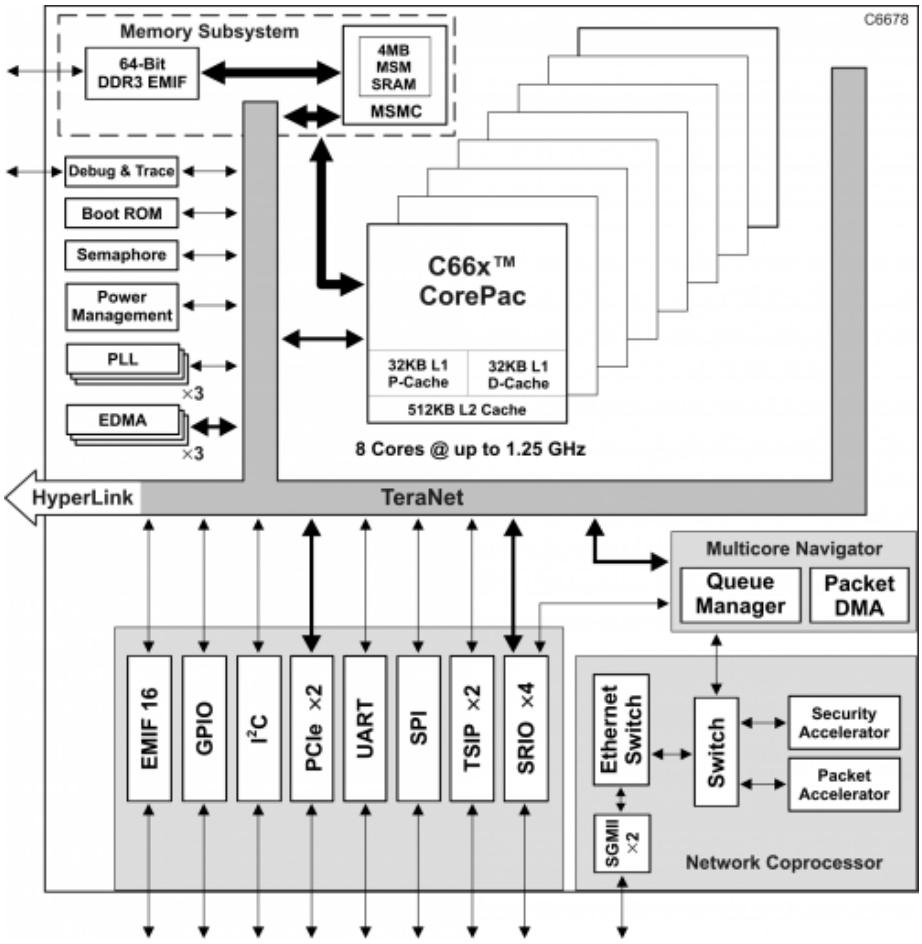


Figure 1.1: C6678 multicore DSP block diagram [72]

The C6678 has a total of 8832KB of on-chip memory organized in multiple levels. Each core has 32KB of level 1 program (L1P) and 32KB of level 1 data (L1D) memory. In addition, each core also has 512KB of local level 2 (LL2) unified high-speed memory, which can be configured either as scratchpad, cache, or a combination

Table 1.1: Memory read performance of the C6678

| | **L1P** | **L1D** | **LL2** | **SL2** | **SL3** | **DDR** |
|---|---|---|---|---|---|---|
| latency (cycles) | ~1 | ~1 | ~7 | ~20 | ~22 | ~80 |

of both. Furthermore, another 4MB of multicore shared memory and 128 KB of ROM are also available to be shared by the eight cores. Portions of the on-chip shared memory may be configured as a shared level 2 RAM (SL2) or a shared level 3 RAM (SL3). SL2 RAM is cacheable only within local L1P and L1D caches, not LL2. SL3 RAM is cacheable by both L1P/L1D and LL2 cache. The ROM contains software used to boot the device. When configured as cache, LL2 memory is always coherent with the L1D cache. However, the shared memory (SL2 and SL3) is not guaranteed by the hardware to be coherent with L1 and L2 caches. Cache coherency must, therefore, be controlled in software to protect the integrity of data that is shared among the cores. Table 1.1 shows the approximate latency, in CPU cycles, for the various memories when a core tries to read a single value from memory.

Several key hardware components facilitate highly efficient and seamless inter-device and intra-device communication in the C6678 [72]. A 64-bit double-data-rate 3 (DDR-3) external memory interface (EMIF) running at 1600MHz supports fast access to external memory while a non-blocking switch fabric called TeraNet provides the interconnect between the C66x DSP cores, peripherals, and memories, and enables fast and contention-free internal data movement. The multicore shared memory controller (MSMC) provides direct access to shared and external memory without directly drawing from TeraNet's capacity. The Multicore Navigator facilitates and manages communications within the device via a queue manager subsystem and a

packet direct memory access (DMA) subsystem. The queue manager subsystem comprises 8,192 hardware queues and is responsible for accelerating the management of packet queues. The packet DMA subsystem optimizes the packet-based communication of the on-chip cores by practically eliminating all copy operations.

## 1.3 Problem Statement

In the past, embedded programmers have depended on using low-level mechanisms for coordinating parallelism and managing memory. This is typically a daunting task, especially considering that this approach is processor-specific and requires that the process must be redone for each change in the instruction set architecture, or to target different deployment processors.

The use of OpenMP to improve programmer productivity, as they write code for more complex multicore systems, has been well studied and justified [36, 32]. OpenMP extends C, C++, and Fortran with a set of compiler directives to express application parallelism in a high-level fashion. The compiler directives are embedded in the code in the form of pragmas for C and C++ programs. However, OpenMP requires special compiler support in order to translate the annotated code into multithreaded code. A runtime library is also usually necessary to manage the execution of the multithreaded code, performing tasks like creating threads, scheduling work units, and managing synchronizations.

While several commercial and open-source implementations of OpenMP exist [3,

4, 7, 5, 73, 8, 22, 69, 17, 52, 9, 51], as far as we are aware, none of these implementations have been deployed for embedded multicore processors. OpenMP implementations often rely on low-level software components, such as an operating system and thread libraries, for implementing concurrency, memory management, and synchronization. While such components exist in general purpose systems, usually with standard interfaces, embedded software typically run on bare-metal or a real-time operating system (RTOS). One of our goals in this dissertation is providing a robust OpenMP implementation that will support the execution of OpenMP applications on an embedded multicore system.

Processors with memory organizations that allow their sizes and/or mode of operation (i.e., as cache or scratchpad memory) to be flexibly configured also raise important questions such as: (i) Should a given memory region be configured as cache or scratchpad? What is the impact of this choice in terms of programmability? (ii) How easy is it to utilize scratchpad memory? (iii) What is the impact of smaller/larger caches on performance? Are large caches always better? While there is previous work that have explored answers to these questions, they have been based on simulations rather than experiments on actual processors. In this dissertation, we have undertaken an empirical performance study using a representative embedded multicore processor as a way to come up with answers to these questions. This study is timely as the mainstream high-performance computing community is starting to look at embedded systems for solutions to overcoming the looming *power wall* [38]. In addition, power savings may be explored by leveraging the ability to reconfigure the sizes of on-chip memory regions to the minimum required to efficiently run a

program [66, 78, 79].

We stated in the previous section that applications with good locality of memory references will usually benefit from utilizing on-chip SRAM. We also mentioned that scratchpad memories typically offer better performance than caches but that their contents must be explicitly managed in software either by a programmer or a compiler. We illustrate this using the following matrix multiplication code:

```
1 for (i = 0; i < N; i++)
2    for (j = 0; j < N; j++)
3       for (k = 0; k < N; k++)
4       {
5          C[i*n+j] += A[i*n+k] * B[k*n+j];
6       }
```

Figure 1.2: Matrix multiplication

We assume that the arrays are stored in a row-major layout. In this code, each element of C is computed using a row of A and a column of B. For simplicity, we will focus only on the spatial reuse of A and B and ignore the fact that the code exhibits limited temporal reuse. Since items are fetched into caches in terms of complete cache lines, some references to A will not hit in the cache if N is large relative to the cache size. There will be much less cache hits for B since it is accessed in a column-wise fashion.

On the other hand, if we were to utilize scratchpad memory for the code, instead of cache, then we could copy the entire row of A and the entire column of B into

scratchpad memory. Subsequently, all references to A and B will be immediately satisfied by scratchpad memory. The code will however need to be re-written to include explicit allocation and copies into scratchpad memory, as well as re-writing the loop to reflect the correct addresses of A and B in scratchpad memory (Figure 1.3).

```
1  //localHeap is a section placed in scratchpad memory
2  newA = Memory_alloc(localHeap,  n*sizeof(dataType), ...);
3  newB = Memory_alloc((localHeap, n*sizeof(dataType), ...);
4  for (i = 0; i < n; i++)
5  {
6     //copy row i of A  into newA
7     dmaInitiateXfer(newA, &matA[i*n], ...);
8     for (j = 0; j < n; j++)
9     {
10      //copy column j of B into contiguous newB
11       dmaInitiateXfer(newB, &matB[j],...);
12       for (k = 0; k < n; k++)
13       {
14       //references to A and B are replaced to reflect new address
             in scratchpad memory
15          C[i*n+j] += newA[k] * newB[k];
16       }
17    }
18 }
```

Figure 1.3: Matrix multiplication using scratchpad memory

We compare the performance of the codes in Figure 1.2 and Figure 1.3 on the C6678 DSP running at 1GHz. The processor has two levels of on-chip memory: 32KB of level 1 memory and 512 KB of level 2 memory. Both memories may be configured as either cache or scratchpad memory. In our experiments, we disabled level 1 memory and measured the number of cycles for the codes in Figure 1.2 using

the cache configuration for level 2 memory, and Figure 1.3 using the scratchpad memory configuration. We measured performance as the number of cycles used to complete the computation and the results are presented in Figure 5.6. While both implementations of matrix multiplication are quite inefficient, it is clear from the figure that the version that utilizes scratchpad memory outperforms the version that uses cache because of the absence of cache misses. Note however, that the data copying code is actually hardware-specific and requires knowledge about details (such as the size and availability of space) of the scratchpad memory.

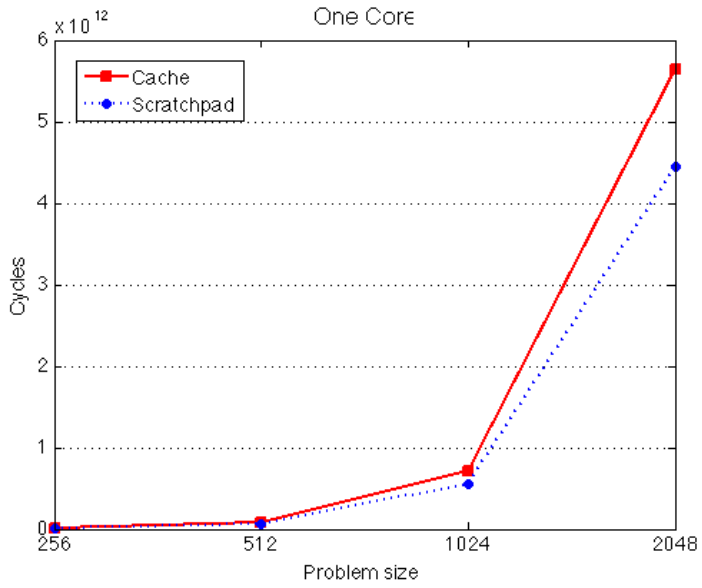


Figure 1.4: Cache vs. scratchpad performance of double-precision matrix multiplication

An OpenMP implementation that supports parallelism and is able to automatically manage scratchpad memory utilization will substantially improve the productivity of programmers and allow for the creation of efficient and portable programs.

## 1.4 Dissertation Overview and Organization

In this dissertation we focus on the problem of implementing OpenMP for embedded multicore processors. The processor representative of this type that is used for this work is the TMS320C6678 (also referred to as C6678) digital signal processor (DSP) manufactured by Texas Instruments.

The primary contributions of this dissertation are the following:

- A compiler implementation of OpenMP within the TMS320C6000 compiler, the production compiler that generates object code for the TMS320C6x class of digital signal processors, which includes our target C6678 processor.
- An empirical performance study of the impact of various memory organizations on the performance of applications running on our representative embedded multicore processor.
- An OpenMP-based infrastructure for automatically utilizing on-chip scratch-pad memory without additional programmer intervention.

The remainder of this dissertation is organized as follows. Chapter 2 reviews shared memory parallel programming models and introduces concepts in OpenMP that will make it easy to follow the rest of this dissertation. Chapter 3 presents our implementation of OpenMP within Texas Instrument's C6000 compiler. Chapter 4 explores the impact of various memory organizations on the performance of applications on our target multicore platform. Chapter 5 details our approach for

OpenMP-based scratchpad memory management. Chapter 6 concludes this dissertation with relevant observations and directions for future work.

# Chapter 2

# Shared Memory Programming Models

In the shared memory model, the underlying hardware is assumed to be a collection of processors or cores, each with access to the same shared memory. Because the cores/processors have access to the same shared memory locations, processors can interact and synchronize with each other through shared variables [44]. This is in contrast to message-passing paradigms that allow parallel programs to be written for distributed memory systems. Message-passing programming models assume that the computing infrastructure is composed of multiple nodes with distinct memory address spaces connected through a communication network. Each compute node can only directly reference its own memory. Communication, of both data and intention, must occur through discrete messages sent from process to process. The message-passing model is generally considered to be quite cumbersome for programmers because they

bear the responsibility for orchestrating all interprocessor communication via explicit messages.

Most shared memory parallel programming models are based on the thread-based model of concurrency called multithreading [48]. We will define two terms—process and thread—before we explain multithreading. A *process* is created by the operating system as a set of physical and logical resources to run a program. A *thread* is the execution state of a program instance. It is an independent stream of instructions that can be scheduled to run. A thread has its own stack, but unlike a process, a thread shares its address space with other threads executing within the same process [12]. Threads belonging to the same process can, therefore, access the same global variables, same heap memory, and the same set of file descriptors. Threads may also have a thread-specific data area for storing data that is not to be accessed by other threads. Multithreading provides the programmer with a means to express concurrency with threads. On a shared memory parallel architecture, each thread can run on a separate processor/core at the same time resulting in parallel execution.

Since threads can access globally shared data belonging to the process, mechanisms must be in place to synchronize thread access to shared data. In order to improve performance, a programming model may also specify a relaxed memory consistency model [11] where each thread can have its own temporary view of shared data. In this model, threads can cache shared data to reduce access times and data consistency is only required at certain points during program execution

In the following, we introduce some of the popular shared memory programming models. We provide the most details on OpenMP since this dissertation is largely

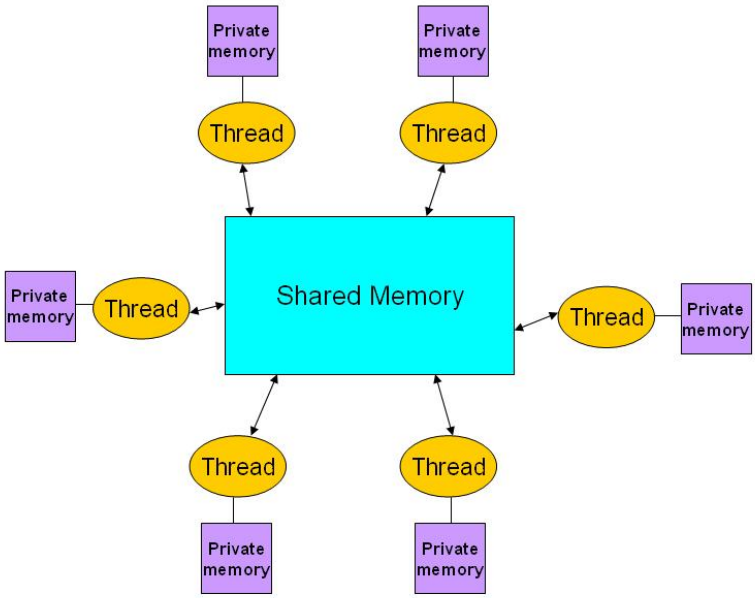


Figure 2.1: Shared memory model of parallel computation

based on it.

## 2.1 POSIX Threads

Portable Operating System Interface (POSIX) threads, commonly referred to as Pthreads, provide a standardized model that implements the POSIX multithreading interface specified by the IEEE POSIX 1003.1c standard [37]. Historically, thread implementations have varied from one hardware vendor to another since each vendor implements its own proprietary versions of threads. This made it difficult for programmers to develop portable multithreaded applications. The Pthreads model was, therefore, created in 1995 as a portable, vendor-neutral solution to this problem.

The Pthreads library provides low-level interfaces for creating and destroying threads and for coordinating thread activities. A multithreaded application typically starts with a single thread. Additional threads may subsequently be created to execute concurrent tasks. This concurrency translates to parallelism on parallel hardware such as multicore processors, since several threads can run on different cores at the same time. Threads can access globally shared data belonging to the process, but the programmer is responsible for synchronizing access to shared data using the Pthreads API.

In general, thread libraries, including Pthreads, may be implemented either in kernel space or in user space [57]. Kernel-space implementations of Pthreads require an operating system with Pthreads support and require system calls to execute most thread operations (e.g., creation, synchronization), making the operations fairly expensive. In addition, the threads are scheduled by the operating system scheduler. User-space implementations of Pthreads, on the other hand, implement thread operations and scheduling entirely in user space. Therefore, in such implementations, thread operations are much cheaper and scheduling can be more flexible.

Since the programmer has explicit control over threads, he must decide how to decompose the application into parallel tasks and create (and destroy) threads to support the decomposition. He must also explicitly control synchronizations to ensure that dependencies are correctly managed. This becomes tedious and error-prone, especially for large applications. Also, since Pthreads are implemented as a library, they do not require compiler support and therefore do not easily facilitate compiler optimizations.

## 2.2 Cilk

Cilk is a multithreaded parallel programming language developed at MIT in 1994 as a parallel extension to C [20, 21, 31]. The main idea behind Cilk is that the programmer focuses on structuring the application in a way that exposes parallelism and data locality, while the Cilk runtime system manages deals such as scheduling, communication, and load-balancing of parallel work among cores [31].

The Cilk language provides a small number of keywords to the programmer to specify parallelism and synchronization. The `cilk` keyword identifies a procedure as a *Cilk procedure* - an asynchronous unit of work. Parallelism is created when the `spawn` keyword precedes the invocation of a Cilk procedure. A Cilk procedure may be considered as a thread or task - a unit of work that can be scheduled. Unlike regular C functions calls where called functions must be completed before the parent function continues, Cilk allows the parent to continue so that it may be executed in parallel with the child. The `sync` keyword is used to force a parent task to wait for its child tasks to complete. Cilk procedures have implicit syncs at the end of the procedure, ensuring that all children terminate before the procedure returns.

The Cilk runtime system uses a work-stealing scheduler under a work-first principle [20, 31]. Workers are located on each processor/core, and each worker maintains a double-ended queue or *deque* for storing ready work. Work is inserted and removed from either end of the deque. Cilk's work-stealing scheduler allows an idle worker to steal work from another worker's deque. Each worker treats its deque as a stack but other workers' deques as queues, ensuring minimal contention for access

to each deque. Cilk's work-first principle forces a newly created Cilk procedure to be immediately executed by a worker while the parent procedure is suspended.

Cilk++ [49], a commercial version of Cilk that supports both C and C++, is distributed by Intel.

## 2.3 Threading Building Blocks

Threading Building Blocks (TBB) is a C++ template library developed by Intel for the programming of multithreaded applications [67, 43]. The library abstracts native threading packages (e.g., Pthreads) with concurrent data structures and parallel algorithms, allowing the definition of tasks which may be dynamically scheduled to cores by the library's runtime system.

TBB provides low-level interfaces for specifying and launching tasks with its `task` and `task_group` APIs. The C++ templates provided by TBB allow programmers to annotate common parallelism patterns such as loop parallelism, parallel reductions, data-flow patterns, and parallel sort. Tasks can then be automatically extracted from the specified patterns.

Tasks may communicate via shared memory through provided concurrent data structures such as concurrent queues, hash maps, and vector. Concurrent structures in TBB are thread-safe structures that are based on fine-grained locking or lockless algorithms. TBB also includes synchronization primitives and memory allocators.

At the core of the TBB runtime library is a task scheduler that automatically

creates and manages a thread pool, thereby hiding the complexity of explicitly managing operating systems threads [68, 24]. The scheduler dynamically assigns tasks to threads and ensures that work is balanced among threads in the pool using a work-stealing algorithm similar to Cilk [68].

Similar to other library-based programming models, TBB does not require special compiler support.

## 2.4 OpenMP

OpenMP is the de facto standard for parallel programming on shared memory multicore systems [6]. It is a specification for a collection of compiler directives, library routines, and environment variables that may be used to specify and control parallelism in C, C++ and Fortran programs. OpenMP offers the programmer full control over parallelization and is therefore classified as an explicit parallel programming model.

OpenMP directives are embedded in programs in the form of pragmas for C and C++ programs, and as sentinels for Fortran programs. OpenMP compiler directives express the parallelism that is to be exploited in a computation as well as necessary synchronizations. The directives are used to express information such as which regions of the program may be executed in parallel by multiple threads, what kind of parallelism exists within specific code regions, and how the work within such regions should be distributed among threads. The library routines and environment variables are used to access or adjust execution parameters of applications such as the

number of threads to be used, thread bindings to cores, and default loop-scheduling options.

### 2.4.1 Execution Model

An OpenMP program begins as a single thread of execution called the *initial thread*. The initial thread creates additional threads, called *slave* or *worker threads*, when parallelism is encountered. The thread that encountered the parallelism becomes the *master thread*, and together the *team of threads*, comprising the master and worker threads, executes the parallel code.

The `parallel` directive is the fundamental OpenMP directive that initiates parallel execution [61]. It marks a structured block of code that should be executed in parallel by multiple threads. When the master thread encounters the `parallel` directive, it generates a set of identical *implicit tasks*, one for each thread, from the code in the associated structured block. Each thread in the team immediately executes its implicit task. *Work-sharing* directives - `for`, `sections`, and `workshare` - express data parallelism and limited task parallelism within parallel regions. They are used to distribute the execution of associated code regions among members of the thread team that encounter the directives. Note, however, that the actual splitting and distribution of the work is handled by the OpenMP implementation. The programmer may direct and tune the work distribution with additional clauses on the directives. More flexible task parallelism is also supported in OpenMP. The `task` directive is used to specify *explicit tasks* - asynchronous units of work that may be

dynamically scheduled for execution by the OpenMP runtime library. An explicit task may be executed immediately once it is created, or its execution may be delayed until some other time during execution of the program. *Synchronization* directives are also provided to coordinate threads' execution and data accesses within parallel regions.

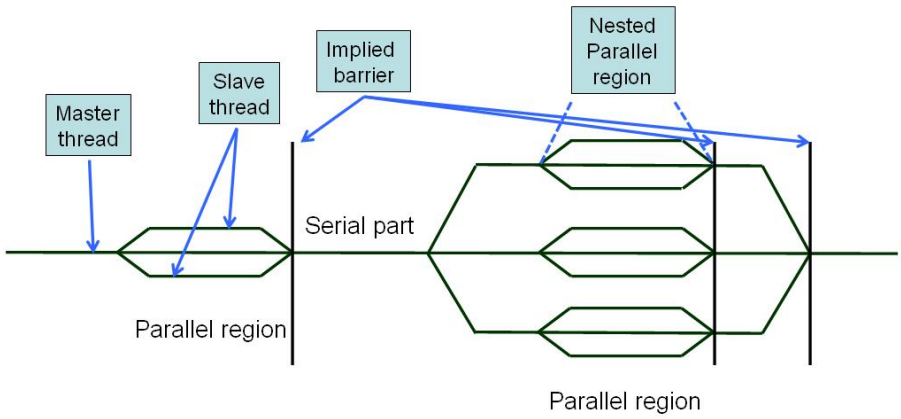


Figure 2.2: OpenMP's fork-join model

At the end of the parallel region, the additional threads join, and only the master thread continues execution of the program. OpenMP, therefore, uses a fork-join model of parallel execution [61] as illustrated in Figure 2.2. Each horizontal line represents a thread of execution. At the start of the first parallel region, the master thread creates two additional threads and, together with these additional threads, forms a team of three parallel threads. At the end of the first parallel region, the slave threads disappear and the master thread resumes execution. Multiple levels of parallelism may be achieved with nested parallel regions. The second parallel region in Figure 2.2 has a nested parallel region. At the start of the nested parallel region,

each thread in the executing team creates a new thread team.

### 2.4.2 Memory Model

OpenMP threads execute in the same shared address space and can share variables within this address space [61]. OpenMP specifies a relaxed-consistency shared memory model where each thread can have its own temporary view of shared data. In this model, threads can cache shared data to reduce access times and data consistency is only required at certain points during program execution. In addition to threads sharing access to variables, OpenMP also allows variables to be designated as private to each thread rather than shared among all the threads. Data-sharing and data-copying clauses used in conjunction with OpenMP directives are used to specify the data environment and determine whether a variable is shared or private. This memory model is similar to the one depicted in Figure 2.1.

### 2.4.3 Constructs

OpenMP constructs form the core elements of OpenMP. The constructs are used to express thread creation, work distribution among threads, data environments, and thread and data synchronization. A summary of the main OpenMP constructs is presented in Table 2.1.

Table 2.1: Summary of commonly used OpenMP constructs

| **Construct** | **Description** |
|---|---|
| parallel | forms a team of threads and starts parallel execution |
| loop | specifies that the iterations of loops will be distributed among and executed by the team of threads that encounter the construct |
| sections | specifies that the enclosed set of structured blocks are to be distributed among and executed by the team of threads that encounter the construct |
| single | specifies that the associated structured block will be executed by only one thread in the team |
| task | defines an explicit task |
| master | specifies that the associated structured block will be executed by only the master thread of the team |
| critical | restricts execution of the associated structured block to a single thread at a time |
| barrier | specifies an explicit barrier at the point in which the construct appears |
| taskwait | specifies a wait on the completion of child tasks of the current task |
| atomic | ensures that a specific storage location is updated atomically |
| flush | enforces a thread's temporary view of memory to be consistent with memory and enforces an order on the memory operations |
| threadprivate | specifies that variables are replicated, with each thread having its own copy |

### 2.4.4 Compiler and Runtime Support

OpenMP compiler directives are embedded in the program in the form of pragmas for C and C++ programs and as sentinels for Fortran programs [61]. Directives for C and C++ begin with #pragma omp, while directives for Fortran begin with !$omp. This well-defined prefix for OpenMP directives makes it easy for compilers to simply ignore the directives and compile the code for sequential execution when OpenMP compilation is not desired, or when the compiler does not support OpenMP.

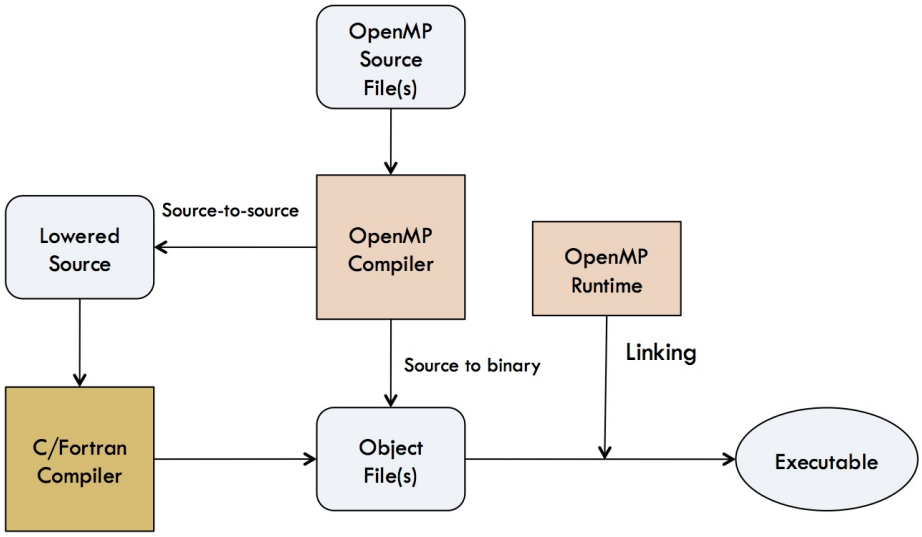


Figure 2.3: Compilation framework for OpenMP

A general compilation framework for OpenMP is shown in Figure 2.3. An OpenMP compiler accepts the source program containing the OpenMP directives and generates multithreaded code that includes calls to a custom OpenMP runtime library. The OpenMP runtime library manages the execution of the multithreaded

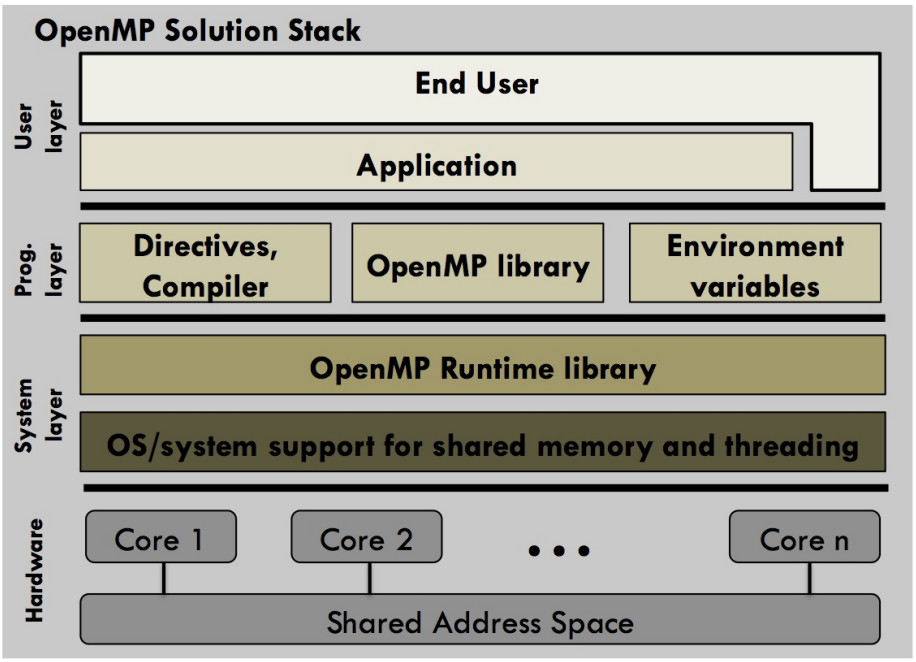


Figure 2.4: Generic OpenMP solution stack

program and is responsible for (i) dynamically creating threads during program execution, (ii) dynamic creation of asynchronous tasks, (iii) scheduling units of work - including tasks, loop chunks, and sections of code - onto threads, (iv) managing any required thread and data synchronizations, and (v) implementing the runtime routines defined in the OpenMP specification. An OpenMP runtime library often relies on lower level components such as the operating system, hardware libraries, and thread libraries in order to complete these tasks. Figure 2.4 shows the interaction between components of a generic OpenMP solution stack. The end user uses compiler directives and library routines to express parallelism in the application. The user could also specify execution behavior by setting environment variables. The compiler lowers the OpenMP directives into multithreaded code with calls to the OpenMP

runtime library. The runtime library manages the execution of the application on the hardware with help from underlying system support for threading and shared memory.

## 2.5 Summary

In this chapter, we introduced shared memory programming models and provided an overview of the main concepts of OpenMP. At its very core, OpenMP allows the parallelization of applications through the use of high-level language constructs.

However, it is the implementation of OpenMP, not just the concepts, that makes all the difference with respect to the performance of applications that are parallelized using OpenMP. An implementation must make decisions about how to split tasks between the compiler and the runtime library, the appropriate compiler translation strategy, and how to implement runtime support for the constructs, while ensuring that implementation overheads are kept to a minimum.

In previous work [46], we highlighted some of the design considerations for implementing OpenMP tasks in a runtime library and studied the impact the choices have on the performance of applications. We also presented microbenchmarks for measuring the implementation overheads of OpenMP tasking constructs [47]. There have been recent efforts to extend the applicability of OpenMP to heterogeneous architectures, where cores have different instruction set architectures and may not have access to a common shared memory. Such heterogeneous architectures are common in the embedded systems domain. Our work in [14] provides a brief discussion on

supporting OpenMP constructs using recent standard low-level interfaces.

# Chapter 3

# OpenMP Implementation in C6000

The OpenMP interface is easy to use and portable across platforms in the sense that the directives provide an abstraction that hides the details of achieving parallelization of an application on a platform from the user. The task of translating an OpenMP application into multithreaded code falls on the OpenMP implementation. Therefore, the quality of the implementation has a dramatic impact on the performance of an OpenMP application.

We introduced a generic OpenMP implementation framework in Section 2.4.4 with a discussion about the role of the compiler and runtime library. The OpenMP specification does not prescribe a specific interface between the compiler and the runtime library; it is up to an implementation to decide the exact roles that each plays in order to generate and support the execution of multithreaded code. A compiler's implementation is, therefore, traditionally tightly coupled with a given runtime library's interface.

In this chapter, we present our OpenMP implementation in the C6000 compiler. We start with a brief overview of the C6000 compiler infrastructure before we go into details about the OpenMP implementation. Next, we present an evaluation of the implementation and briefly discuss other OpenMP implementations. Finally we conclude the chapter with a summary.

## 3.1 The C6000 Compiler

The TMS320C6000 C/C++ compiler is an optimizing compiler that belongs to, and is maintained by, Texas Instruments. The compiler accepts C and C++ code conforming to the International Organization for Standardization (ISO) standards for these languages, performs a wide variety of optimizations to improve the efficiency of the code, and produces object code for the TMS320C6x set of DSPs produced by Texas Instruments [71].

The compiler consists of separate tools that are invoked in independent phases of the compilation process. The *shell* drives the overall compilation processes and invokes each tool in order, and with the desired options. The *parser* is based on the Edison Design Group (EDG) [2] front end, and reads the C/C++ source file, performs preprocessing functions, verifies the syntax, and produces an intermediate file. The *optimizer* reads the intermediate file generated by the parser, and performs various general and target specific optimizations to improve the execution speed and/or size of the program. The *code generator* converts the intermediate file generated by the parser or the optimizer to an assembly language source code for the TMS320C6x

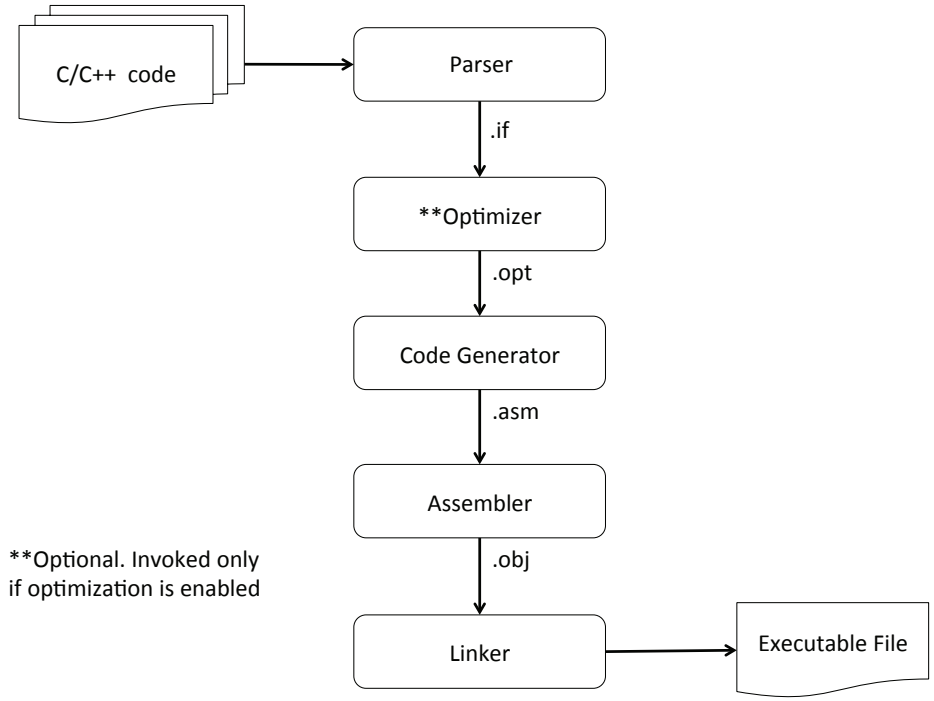


Figure 3.1: Layout of C6000 compiler framework

device. The assembler translates assembly language source code files into machine language object files. The *linker* combines object files into a single executable output file. It also allocates relocatable sections and symbols and resolves external references between input files. The layout of the generic compiler framework is shown in Figure 3.1.

## 3.2 OpenMP Support in C6000

OpenMP support for C in the C6000 compiler comprises three main components. The first component involves parsing/recognizing the OpenMP constructs. The second aspect involves translating the OpenMP constructs into multithreaded code with calls to a custom runtime library. The third component is the runtime library implementation that facilitates the execution of the multithreaded code. Our focus in this dissertation was on the first two components, while a separate team at Texas Instruments worked on the runtime library component.

### 3.2.1 Parsing OpenMP Constructs

The EDG front end reads C source files and produces an intermediate language tree that represents the source program. We modified the EDG front end to recognize OpenMP constructs and include these constructs in the intermediate language.

Our starting point was the OpenMP C context-free grammar included in the OpenMP specification [61]. We converted it to a Yacc grammar [40, 41] that included Lex rules to recognize the OpenMP tokens. Then we used GNU Bison [1], a general-purpose parser generator, to convert the grammar into a C program that parses the OpenMP constructs. The EDG tokenizer is used to grab tokens from the C source files. Once an OpenMP pragma token is encountered, the pragma is parsed using the generated grammar. A record is created for the pragma and the pragma is then associated with a statement record within the intermediate language tree.

### 3.2.2 OpenMP Translation

We previously mentioned that a compiler implementation typically targets a specific runtime library interface. The GNU OpenMP (GOMP) [58] runtime interface was chosen as the target for our OpenMP compiler implementation. As we mentioned before, a separate team customized the runtime library to use TI's RTOS called SYS/BIOS and other custom modules for inter-process communication.

#### 3.2.2.1 Parallel and Task Constructs

OpenMP's parallel and task constructs define regions of code that correspond to implicit and explicit tasks respectively, and may be concurrently executed by a thread team. Our translation strategy for these constructs was to outline the body of parallel and task regions into separate functions. The outlined functions are subsequently used as arguments to the runtime library's thread creation or scheduling routines.

During outlining, the data-sharing attribute—shared or private—of each variable in the region is determined. An activation record structure is created, which each field of the structure specifying the address of shared variables. New local variables are created within the outlined function for each private variable. As statements are copied from their initial positions within the body of the parallel region to the newly created outlined function, shared and private variables are remapped accordingly. Private variables are remapped to their corresponding local variables, while shared variables are remapped to their corresponding fields in the activation record structure. Finally, the parallel and task regions are replaced with calls to appropriate

runtime library routines to either immediately launch the execution of the functions, for parallel regions, or possibly defer their execution to some other time, for explicit tasks.

We illustrate the translation of the parallel construct with a simple OpenMP program in Figure 3.2. Each thread in the `parallel` region decides what part of the global array $x$ to work on, based on its unique thread number.

```
1  float x[10000];
2     int iam, nt, istart, ipoints, i;
3     int npoints = 10000;
4  #pragma omp parallel private(i, iam, nt, istart, ipoints)
5     {
6        iam = omp_get_thread_num();
7        nt = omp_get_num_threads();
8        ipoints = npoints / nt;
9        istart = iam * ipoints;
10       if (iam == nt-1)
11          ipoints = npoints - istart;
12       for (i = 0; i < ipoints; i++)
13          x[istart+i] = 123.456;
14    }
```

Figure 3.2: Simple OpenMP program. Threads' unique IDs are used decide what section to work on.

Figure 3.3 shows the resulting translated code. The compiler generates a new function named *__omp_fn_0* that contains the code that each thread will run (i.e., the code within lines 5 – 14 of Figure 3.2). The new *temp_12* structure variable is created and is used to store the addresses of shared variables $x$ and *npoints*. Also, new private variables are created within *__omp_fn_0*, and now replace the variables that are

```
    ...
    auto struct omp_data_s5 temp_12;
    ...
    (temp_12.x) = (&x);
    (temp_12.npoints) = (&npoints);
    GOMP_parallel_start(__omp_fn_0, ((void *)(&temp_12)), ((!(1U))
       ? 1U
 : 0U));
    __omp_fn_0(((void *)(&temp_12)));
    GOMP_parallel_end();
    ...

static void __omp_fn_0(
void *__T141030200)
{
  auto int temp_11;
  auto int temp_10;
  auto int temp_9;
  auto int temp_8;
  auto int temp_7;
  auto struct omp_data_s5 *omp_data_p6;

  omp_data_p6 = ((struct omp_data_s5 *)__T141030200);

  temp_10 = (omp_get_thread_num());
  temp_9 = (omp_get_num_threads());
  temp_7 = ((*(omp_data_p6->npoints)) / temp_9);
  temp_8 = (temp_10 * temp_7);
  if (temp_10 == (temp_9 - 1))
    temp_7 = ((*(omp_data_p6->npoints)) - temp_8);
  for (temp_11 = 0; (temp_11 < temp_7); temp_11++)
    (((*(omp_data_p6->x)))[(temp_8 + _temp_11)]) = ...
}
```

Figure 3.3: Translation for program in Figure 3.2

designated as private within the parallel region. The parallel region is replaced with a call to the runtime library function *GOMP_parallel_start()*, which takes this newly created function, as well as the address of the activation record structure and the number of threads, as arguments. *GOMP_parallel_start* is called by the encountering

thread, and is used to launch threads that will each execute *__omp_fn_0()*. The encountering thread becomes the *master* of the thread team, and proceeds to execute *__omp_fn_0* as well. After completing the execution of this function, the master thread calls *GOMP_parallel_end()*. This runtime library function ensures that all threads reach this point before the master thread proceeds, and puts the thread team to sleep.

#### 3.2.2.2 Worksharing Constructs

Worksharing constructs distribute the execution of the associated region among the members of the encountering team of threads. These constructs usually exist within the implicit tasks created from parallel regions, but each thread executes only a portion of the work enclosed by the worksharing construct.

Since these constructs exist within parallel regions, there is no need to outline them into separate functions. If a worksharing construct exists within the lexical scope of its enclosing parallel region, then the region also gets outlined into the new function that represents the implicit task. However, if the worksharing construct is not within the lexical scope of its enclosing parallel region, it will simply be invoked during the execution of the outlined function.

Most tasks associated with distributing work to threads are deferred until execution and are handled by the runtime library. Translation of the constructs mainly involved introducing the runtime routines for parceling work to threads into the code,

and, for the `omp for` construct, rewriting loop bounds as variables that are determined by the runtime routines. An exception to this strategy is translating the `omp for` with the *static* schedule when no chunk size is specified. In this case, loop iterations are divided into chunks that are approximately equal in size, and each thread executes at most one chunk. It becomes unnecessary to defer the work distribution until runtime; the compiler can achieve this task with only limited runtime calls.

```
1 #pragma omp for
2    for (i=1; i<n; i++) /* default loop schedule is static */
3       b[i] = (a[i] + a[i-1]) / 2.0;
```

Figure 3.4: Simple parallel loop with static schedule.

```
1 #pragma omp for schedule(dynamic)
2    for (i=1; i<n; i++)
3       b[i] = (a[i] + a[i-1]) / 2.0;
```

Figure 3.5: Simple parallel loop with dynamic schedule.

We contrast the translation of `omp for` static and dynamic schedules using Figures 3.4 and 3.5. These are similar programs except that the former uses a static loop schedule for distributing the loop iterations to threads, while the latter uses a dynamic loop schedule. Figures 3.6 and 3.7 are the resulting translations of the codes in Figures 3.4 and 3.5 respectively. When no chunk size is specified for the static schedule, each thread's unique ID and the size of the thread team are

sufficient to compute the upper and lower bound of the loop that each thread will independently execute. For the dynamic schedule, the *GOMP_loop_dynamic_start* and *GOMP_loop_dynamic_next* runtime library calls return the appropriate upper and lower bounds at runtime until all the iterations have been distributed to threads.

```
{
   temp_3 = omp_get_thread_num();
   temp_4 = omp_get_num_threads();
   temp_5 = 1;
   temp_6 = temp_5 + (-1);
   temp_7 = ((n - 1) + temp_6) / temp_5;
   temp_8 = temp_7 / temp_4;
   temp_8 = ((temp_8 * temp_4) != temp_7) + temp_8;
   temp_9 = temp_8 * temp_3;
   temp_10 = temp_9 + temp_8;
   temp_10 = (temp_10 > temp_7) ? temp_7 : temp_10;
   temp_1 = (temp_9 * temp_5) + 1;
   temp_2 = (temp_10 * temp_5) + 1;
   if (temp_9 < temp_10)
   {
      for (temp_0 = temp_1; temp_0 < temp_2; temp_0++)
      {
          b[temp_0] = (a[temp_0] + a[temp_0 - 1]) / 2.0;
      }
   }
   GOMP_barrier();
}
```

Figure 3.6: Translation of static schedule.

#### 3.2.2.3 Other Constructs

Translation of some of the constructs are more straightforward. `omp barrier`, `omp flush`, `omp taskwait` are each replaced with calls to *GOMP_barrier()*, *GOMP_flush()*,

```
1  auto int n = 1000
2  {
3    if (GOMP_loop_dynamic_start(1, n, 1, 1U, (&temp_1), (&temp_2)))
4    {
5       do
6       {
7          for (temp_0 = temp_1; (temp_0 < temp_2); temp_0++)
8          {
9             b[temp_0] = (a[temp_0] + a[temp_0 - 1]) / 2.0;
10         }
11
12      } while (GOMP_loop_dynamic_next((&temp_1), (&temp_2)));
13   }
14   GOMP_loop_end();
15 }
```

Figure 3.7: Translation of dynamic schedule.

and *GOMP_taskwait()* runtime routines, respectively. `omp master` is translated by wrapping the associated statement with an if statement to check that only thread 0 executes the statement.

Translation for `omp critical` and `omp atomic` constructs is achieved by inserting runtime routines at the beginning and end of the associated statement(s).

## 3.3 Evaluation

In this section, we use two case applications – matrix multiplication and conjugate gradient – to demonstrate that our OpenMP implementation improves performance by supporting parallel execution of these programs.

### 3.3.1 Matrix Multiplication

The OpenMP code for matrix multiplication is displayed in Figure 3.8. We measured the performance of this code on the C6678 for a matrix size of 1024 x 1024 while increasing the number of OpenMP threads from one to eight. Figure 3.9 shows that we are able to achieve near-linear speedup, obtaining a speedup up of 6.22 on eight processors.

```
1 #pragma omp for private(i, j, k)
2 for (i = 0; i < N; i++)
3    for (j = 0; j < N; j++)
4       for (k = 0; k < N; k++)
5       {
6          C[i][j] += A[i][k] * B[k][j];
7       }
```

Figure 3.8: OpenMP version of matrix multiplication.

### 3.3.2 Conjugate Gradient

The Conjugate Gradient (CG) application is from the NAS OpenMP benchmarks (NPB), a set of applications designed to evaluate the performance of parallel computers [39]. CG estimates the smallest eigenvalue of a large, sparse, symmetric positive-definite matrix using the inverse iteration coupled with a conjugate gradient method.

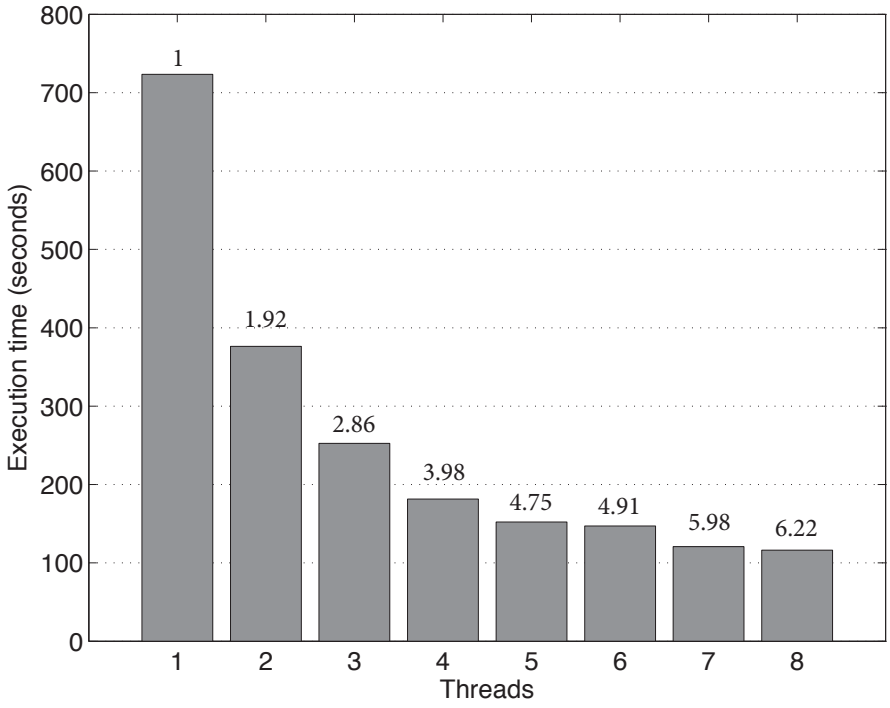


Figure 3.9: Performance of double-precision matrix multiplication. Speedup over one thread execution is included at the top of each bar.

As with matrix multiplication, we measured the performance of CG on the C6678 with increasing number of OpenMP threads. As Figure 3.10 shows, our implementation obtains a speedup of 5.57 for 8 threads.

## 3.4 Related Work

A number of commercial and open source compilers currently support OpenMP. Commercial implementations are typically not accessible to the research community

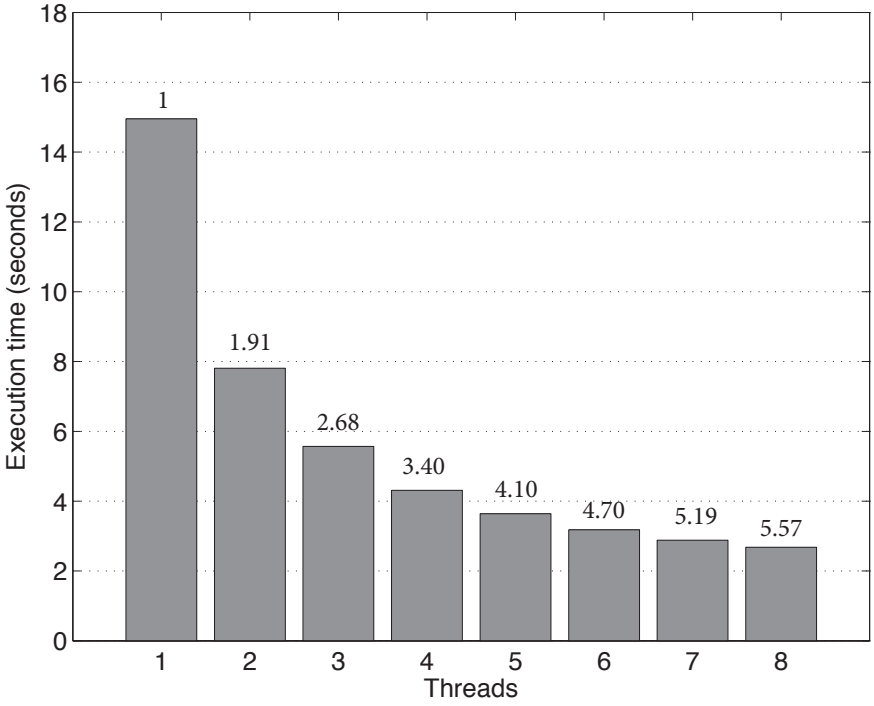


Figure 3.10: Performance of CG. Speedup over 1 thread is included at the top of each bar.

and usually target specific machine architectures. Examples of commercial compilers with OpenMP support include the C/C++/Fortran compilers from IBM [3, 4], Oracle [7], Intel [5, 73], and Pathscale [8].

Open source compilers typically take a source-to-source translation approach where the OpenMP constructs are lowered into multithreaded code with calls to a runtime library. A backend compiler is, however, required to complete the compilation approach and generate executable. These include OdinMP/CCp [22], Omni [69], Nanos Mercurium [17], and ROSE [52]. Exceptions to these are GCC [9] and

OpenUH [51] which provide source to binary translations for OpenMP. Most of these are used as research platforms for exploring topics such as interoperability of OpenMP with other programming models, optimizations for OpenMP programs, as well as language extensions for OpenMP.

OpenMP has also been implemented on other platforms other than general purpose processors. Chapman et al. [23] leveraged the source-to-source translation capability of OpenUH in conjunction with the Texas Instruments C6000 compiler to implement OpenMP for a multicore DSP platform. Their work focused on a new OpenMP runtime library that was customized for the memory organization of the target multicore DSP. He et al. [35] also created an OpenMP compiler and runtime implementation for the MSC8156 six-core DSP processor from Freescale Semiconductors.

## 3.5 Summary

In this chapter, we discussed the basic compiler implementation of OpenMP in the C6000 compiler. We showed that our implementation is able to improve the execution time of applications by generating and supporting the execution of multithreaded code. The sizes and mode of the on-chip memory spaces of the C6678 may also be flexibly configured. In Chapter 4, we will explore the impact of various memory organizations on the performance of parallel applications. In Chapter 5, we will then describe a methodology that allows our implementation to direct the utilization of scratchpad memory.

The work contained in this chapter was carried out at Texas Instruments under the supervision of Dr. Eric Stotzer.

# Chapter 4

# Effects of On-chip Memory Organization on Performance

## 4.1 Introduction

Studies have shown that on-chip memories account for a significant fraction of the overall energy consumed by embedded systems [18, 66, 78, 79]. While desktop and HPC systems typical have fixed memory organizations that cater to a wide range of applications, embedded systems usually allow their memory organizations to be tuned for specific classes of applications. This allows them to exploit optimal performance at lower energy costs [66, 78, 79].

There are two main schemes for implementing on-chip memory: cache and scratch-pad. There are tradeoffs for choosing either scheme. In the following, we investigate

the impact of various on-chip memory sizes and mode of operation (cache or scratch-pad) of the C6678 DSP on the performance of applications.

## 4.2 On-chip Memory Organization

Over time, processor core speed has been increasing at a more rapid rate than memory speed, leading to the *memory wall* problem [77] where the CPU stalls while waiting for memory requests to be fulfilled. John von Neumann recognized this point already in 1946 [75] stating that:

"It shows in a most striking way where the real difficulty, the main bottleneck, of an automatic very high speed computing device lies: At the memory."

A solution to this problem is incorporating static random access memories (SRAM), which are very fast but relatively smaller and more expensive memories, within chips in order to reduce the number of off-chip main memory (DRAM) accesses. The SRAM will contain a subset of a program's dataset and/or instructions for faster access. When the memory reference pattern of an application has good spatial and/or temporal reference locality (i.e., memory references are clustered in space and/or time), they can benefit from having on-chip SRAM. On-chip SRAM is commonly organized in a hierarchical fashion with the fastest but smallest memory level located closest to the CPU, and subsequent levels being slower, but larger and cheaper. The effect of hierarchical organizations is a larger pool of memory that ideally serves data to programs at roughly the rate of the fastest memory at the top of the hierarchy, but costs roughly as much as the cheapest memory at the bottom of the hierarchy.

SRAM blocks may be operated either as caches or scratchpad memory.

### 4.2.1 Cache

A cache serves as an interface between the CPU and main memory. Caches are not addressable; instead the idea is for a cache to temporarily hold frequently accessed data items or instructions so that most memory accesses are fulfilled by the cache instead of main memory (see Figure 4.1). Unified caches contain both data and instructions, while split caches have separate spaces for instructions and data. Caches are organized as a number of *cache lines* and information transfer between the cache and main memory is in terms of complete cache lines, rather than individual bytes.

The *associativity* of a cache determines into which line in the cache a memory location will go. A *fully-associative* cache allows cache lines to be stored at any location in the cache. This scheme offers the best performance because any memory location can be stored at any cache location, but it also requires a large number of address comparisons to determine the presence of a memory location in cache making it rather complex and expensive to implement. An *N-way set associative* cache splits the entire cache into a number of sets, each containing $N$ cache lines. Each memory location is assigned a set and may be cached in any one of that set's $N$ lines. This is less complex to implement than fully-associative caches since only N comparisons are required to determine the presence of a memory location in cache. A *direct-mapped* cache is a 1-way set associative cache; a memory location may only be cached in a single cache line. This is the least complex to implement of all three schemes since

only one comparison is required to check the presence of a memory location in the cache.

When a processor makes a memory request, it first checks whether the relevant item is contained in the cache. A *cache hit* occurs if the requested item is in the cache and the item is read from or written to the cache instead of the much slower main memory. When the CPU does not find the requested item in the cache, it results in a *cache miss* and the item must be fetched from the main memory into cache. A cache replacement policy determines which cache line will be evicted when a new cache line is fetched into a full cache. Common replacement policies include least recently used (LRU) which evicts the least recently accessed cache line from the cache and FIFO which evicts the oldest cache line.

The main advantage of using caches is the convenience that it offers to programmers. The contents of caches are automatically managed by the hardware and most programs can successfully exploit the benefits of caches without any changes to the source code. However, the best performance is achieved when a program is carefully tuned for, or is able to adapt to, a specific cache organization. The main drawback of caches is that they do not guarantee performance predictability; they are subject to compulsory, capacity and conflict misses. Cache performance depends heavily on the history of memory accesses as well as the cache placement and replacement policies employed.

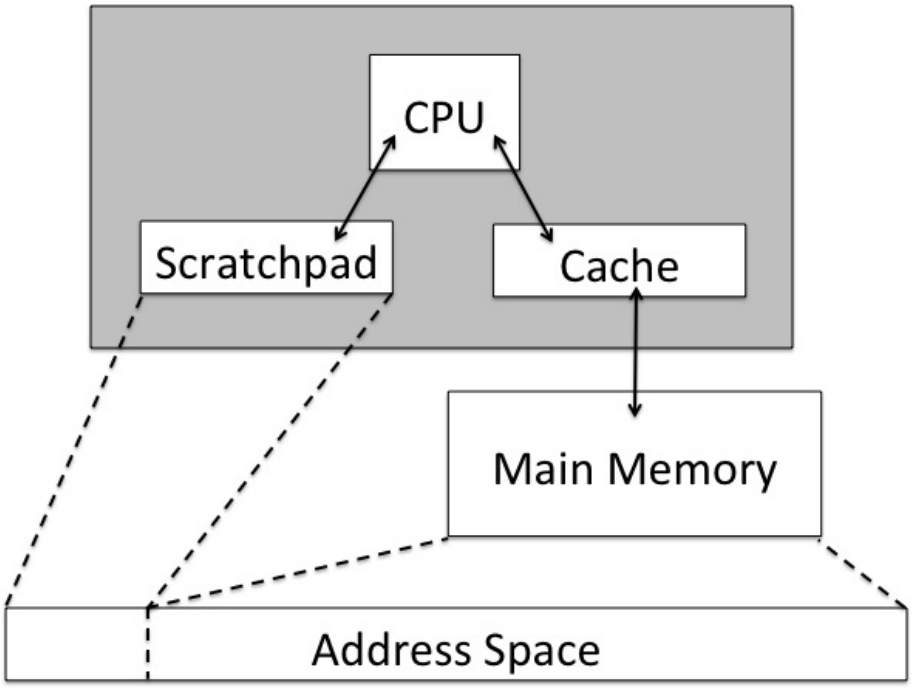


Figure 4.1: Cache versus scratchpad memory

### 4.2.2 Scratchpad

Scratchpad memories are small on-chip SRAMs that are mapped to a pre-determined address space that is disjoint from main memory; they are therefore directly addressable by the CPU. Memory references to addresses within the pre-determined scratchpad address range are served by scratchpad (see Figure 4.1). Memory references outside this window are served by main memory (possibly through a cache). Scratchpad memory is entirely under software control - program or data objects must be explicitly placed there before they are accessed.

Unlike caches, scratchpad offers predictable performance since there is no need

to check for the availability of data/instruction in the scratchpad. This makes it very suitable for real-time embedded systems where deterministic performance is required. In addition, pure scratchpad memories are considerably more energy and area efficient than caches since there is no need for comparator and signal miss/hit circuitry which is used in caches [18]. However, utilizing scratchpad requires explicit allocation (and dynamic movement) of data/instructions either by a programmer or a software tool such as a compiler which requires knowledge about the size of the scratchpad as well as data/instruction access patterns within the application.

## 4.3 Methodology

We introduced the C6678 architecture in Section 1.2. Let us recapitulate the salient features of its memory organization. The C6678 has a total of 8.5MB of on-chip memory organized in multiple levels. Each core has 32KB of level 1 program (L1P) and 32KB of level 1 data (L1D) memory. In addition, each core also has 512KB of local level 2 (LL2) unified memory. Furthermore, another 4MB of multicore shared memory (MSM) SRAM is also available to be shared by the eig ht cores. L1D, L1P and LL2 can each be configured independently either as scratchpad memory, or cache, or a combination of both. In the cache configuration, L1P operates as a direct-mapped cache while L1D operates as a two-way set associative cache. Their cache sizes (0KB, 4KB, 8KB, 16KB, or 32KB) can be selected at runtime. When configured as cache, LL2 operates as a four-way set associative cache and acceptable sizes (0KB, 32KB, 64KB, 128KB, 256KB, 512KB) can be selected and changed at

Table 4.1: Applications used

| Benchmark | Description | Problem Size (class S) | Problem Size (class A) |
|---|---|---|---|
| CG | Estimates the smallest eigenvalue of a large, sparse, symmetric positive definite matrix using inverse iteration with a conjugate gradient method. | 1400 rows | 14000 rows |
| EP | Generates pairs of Gaussian random deviates from a set of uniformly distributed random numbers. | 2ˆ24 random pairs | 2ˆ28 random pairs |
| FT | Solves a 3D partial differential equation using Fast Fourier Transforms (FFT). Three one-dimensional FFTs are performed for each dimension. | 64 x 64 x 64 grid | 256 x 256 x 128 grid |

runtime. Portions of the MSMC memory may be configured as shared level 2 SRAM (SL2) or shared Level 3 SRAM (SL3). SL2 SRAM is cacheable only within local L1P and L1D caches, not LL2, while SL3 SRAM is cacheable by both L1P/L1D and LL2 cache.

The EP, CG, and FT applications from the NAS OpenMP benchmarks (NPB)[39] were selected for our study. Table 4.1 summarizes the key features of these three applications.

### 4.3.1 Experiment Setup

In our evaluations, we measure performance as the time required to complete the computations. For cache-based memory configurations, our goal is to understand the impact of level 2 (L2) and level 1 (L1) cache sizes on the performance of the benchmarks. In this configuration, we set the cache size and made no changes to the source programs. Utilizing scratchpad involved adding code for explicit memory allocations from scratchpad memory. Note that except for EP, the entire datasets are too large to fit in scratchpad memory. For the sake of simplicity, we used a static allocation approach where we chose frequently accessed array structures that will benefit most from being allocated in scratchpad memory.

We executed the programs with 1, 2, 4, and 8 OpenMP threads (each thread runs on a separate core). Reported measurements are the average of three runs of each benchmark for all results. The benchmarks were compiled with version 7.4.0 of the TI C6000 compiler [71].

## 4.4 Results

### 4.4.1 Impact of Level 2 Cache Size

In order to measure the impact of varying L2 cache sizes, we kept the L1D cache size fixed at the maximum size of 32KB while setting the L2 cache size to 0KB, 32KB, 64KB, 128KB, or 256KB. We were unable to set the L2 cache size to 512KB

because a section of the local level 2 memory is allocated for the compiler to place variables that must be local to each core. The performance of the benchmarks for these level 2 cache sizes are presented in Fig. 4.2 for class S, and Fig. 4.3 for class A. Our evaluations reveal that for the smallest problem size, class S, there is very little variation in performance for L2 cache sizes exceeding 32KB. The reason for this is that the size of the dataset allows the majority of the memory requests to be serviced by the caches.

For the larger problem size, Class A (Fig. 4.3), we observe that increasing the level 2 cache sizes reduces the execution time of the application for two of the three benchmarks. The impact of increasing L2 size is most significant in CG, where we observe an average of 71 percent improvement in performance when we change L2 cache size from 32KB to 256KB.

Of the three benchmarks, CG is the most memory-intensive and memory-sensitive; the majority of its execution time is spent on a sparse matrix-vector multiplication. A matrix access involves indirect addressing which prevents spatial reuse of cache lines and causes a large number of L1 cache misses. For larger L2 cache sizes, most of the L1 misses are satisfied by L2. However, as we reduce the size of L2 cache, less L1 misses hit L2 and instead access external memory.

FT shows a similar, though less dramatic, trend as CG with an average of 40 percent improvement in performance when we change L2 cache size from 32KB to 256KB. The benchmark is less memory-intensive than CG and it involves compute-intensive phases that are able to benefit from intensive cache accesses. For the third benchmark, EP, no variation is observed with varying L2 cache size because EP is

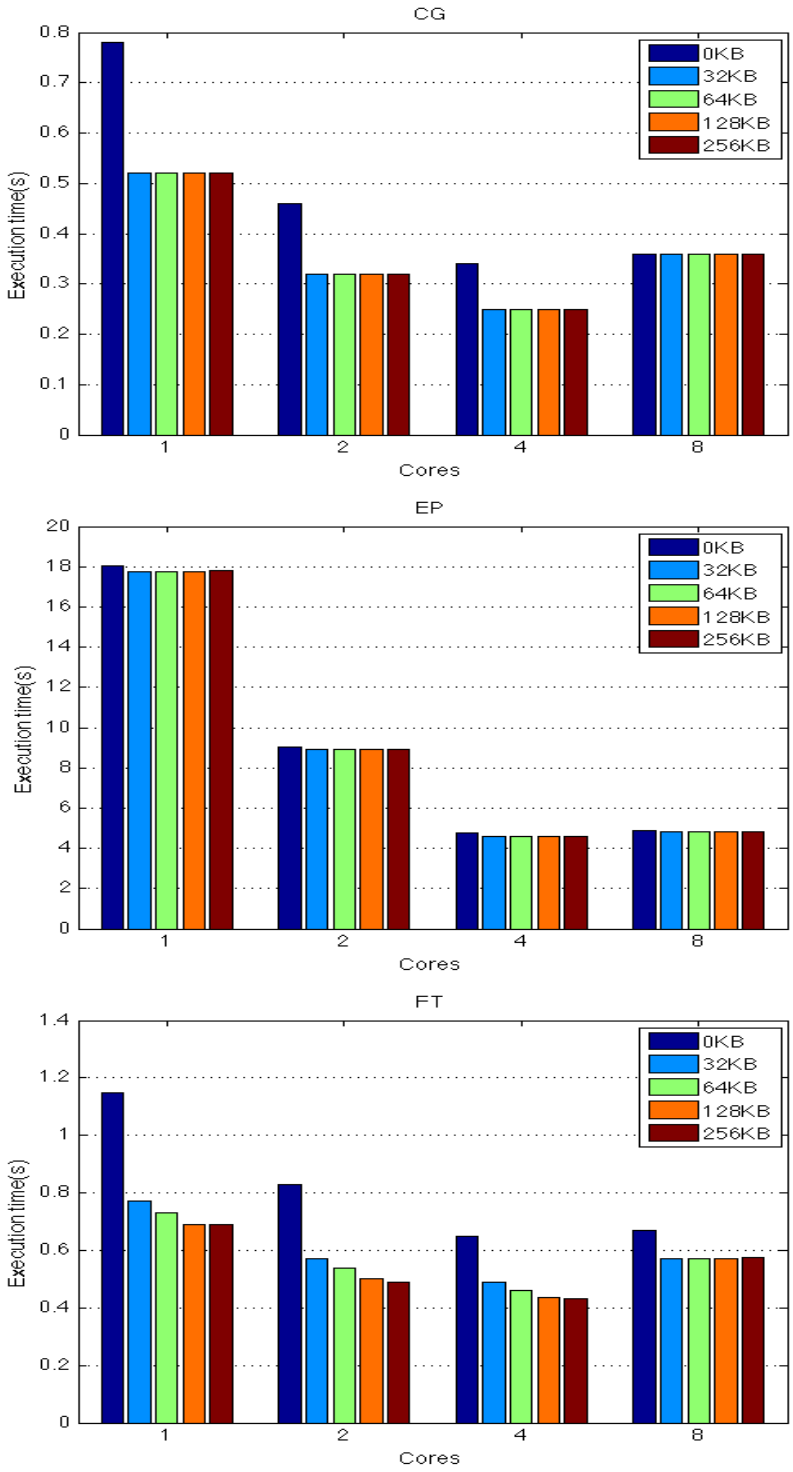


Figure 4.2: Execution times for the NAS Benchmarks Class S for varying L2 cache sizes and number of threads.

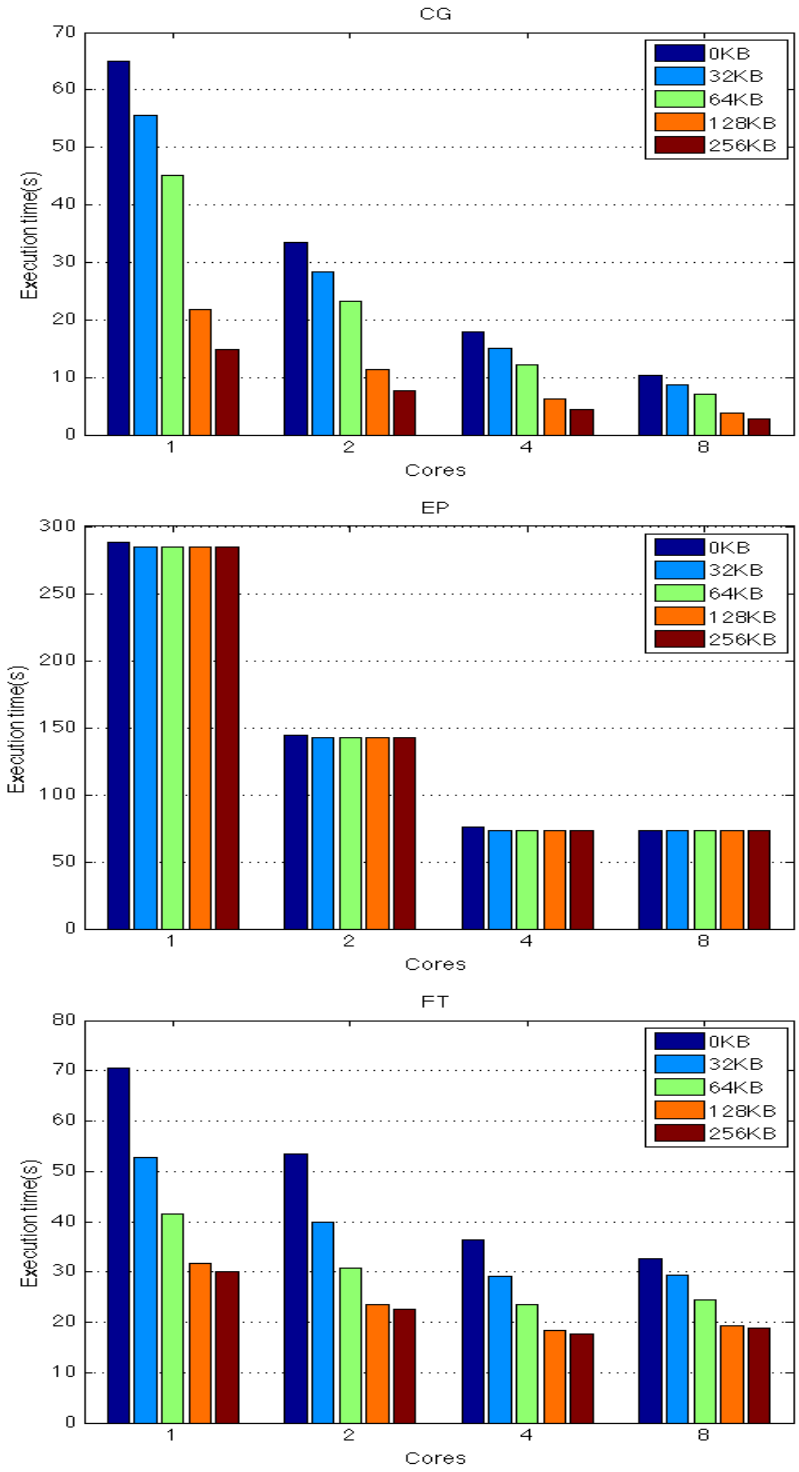


Figure 4.3: Execution times for the NAS Benchmarks Class A for varying L2 cache sizes and number of threads.

a pure CPU-intensive benchmark and its performance is not dominated by memory accesses.

### 4.4.2 Impact of Level 1 Cache Size

We measured the impact of varying L1 cache sizes by keeping the L2 cache size fixed at 256KB while setting the L1 cache size to 0KB, 4KB, 8KB, 16KB or 32KB. The performance of the benchmarks for these L1 cache sizes is presented in Fig. 4.4 for class A. Our evaluations reveal no significant variation in performance for varying L1 cache sizes except when the L1 cache size is set to 0KB. The performance hit when L1 cache is set to 0KB is likely due to the accesses to scalar variables and smaller arrays that the L1 cache had been able to reuse.

### 4.4.3 Utilizing Scratchpad Memory

An application can utilize scratchpad memory either by instructing the compiler to allocate all variables in the scratchpad memory region (option 1) or by including explicit software instructions. These instructions can be either inserted manually by the programmer (option 2), or generated by the compiler (option 3). We note here that the C6000 compiler currently has no support for automatic scratchpad memory utilization. Therefore, we chose either option 1 or option 2 in our evaluations.

The size of the application's dataset determines which option we utilize. We made the decision to use option 1 when the entire dataset fits in scratchpad, and option 2 when we only select a subset of the dataset to be allocated in scratchpad.

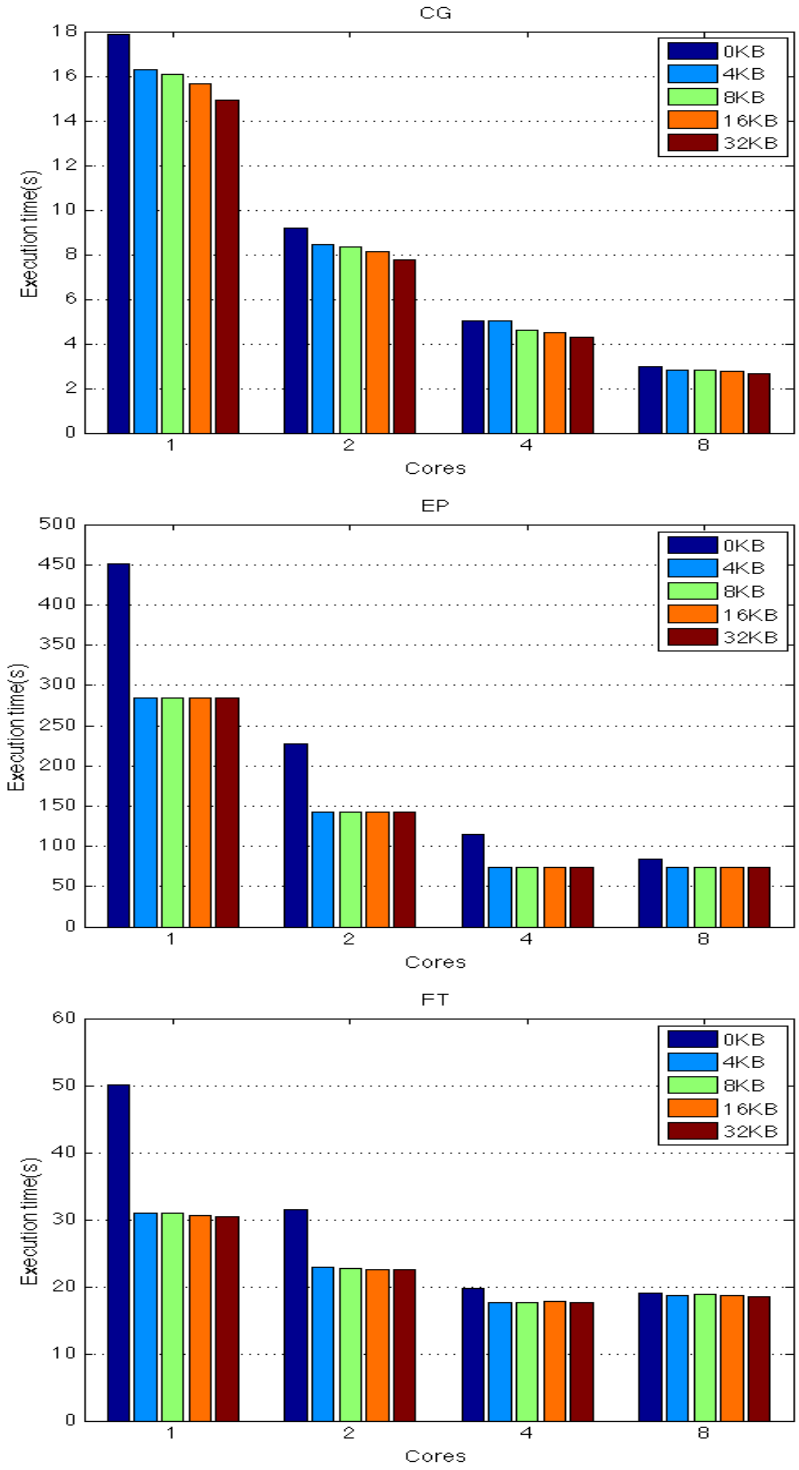


Figure 4.4: Execution times for the NAS Benchmarks Class A for varying L1 cache sizes and number of threads.

For FT, we place both the 1-D FFT input and output arrays in scratchpad memory. For class A, this requires 128KB which we allocate from level 2 memory. Figure 4.5 shows the difference in performance when a pre-determined size of level 2 memory is configured entirely as cache versus using all or part of the level 2 memory as scratchpad. The figure at the top shows the difference in performance when 128KB of level 2 memory is used entirely as scratchpad memory versus using it entirely as cache. The figure at the bottom shows the difference in performance when 256KB of level 2 memory is used entirely as cache versus using half (128KB) as cache and the other half as scratchpad memory. We make two observations from both figures. First, we note that the performance is better when all or part of the memory region is used as scratchpad than when it is used entirely as cache. This is due to the overall reduction in misses for level 2 memory thanks to of our use of scratchpad. Second, we observe that the scratchpad configuration scales up much better than the all0cache configuration when the number of threads increases.

In our experiments with EP, we utilized the scratchpad memory for the array that holds the generated random numbers. We found its performance with the scratchpad configuration to be comparable with its performance with the cache configuration. As we observed earlier, this is because its performance is not dominated by memory accesses.

Table 4.2: Comparison of FT execution time (in seconds) on C6678 and 8-core Opteron

| | C6678 | Opteron |
|---|---|---|
| 1 | 34.64 | 14.42 |
| 2 | 19.11 | 8.37 |
| 4 | 11.41 | 5.53 |
| 8 | 7.72 | 4.24 |

### 4.4.4 Comparison with a Higher-end Multicore Processor

For the sake of comparison, we also measured the performance of FT on a general purpose multicore system with dual 2.2 GHz quad-core AMD Opteron 2354 processors for a total of eight cores. Each core has 64KB L1 and 512KB L2 caches with a shared 2MB L3 cache per processor. Each processor has a power consumption of 75 Watts. The application was compiled with the GNU C compiler 4.6 with optimizations enabled at level -O3. Table 4.2 presents the performance we obtained on this processor and on the C6678 (version of the FT that used 256KB of L2 memory with half scratchpad memory and half cache). As we can see, the much cheaper C6678 is able to achieve about half the performance of the Opteron while using only 10 Watts.

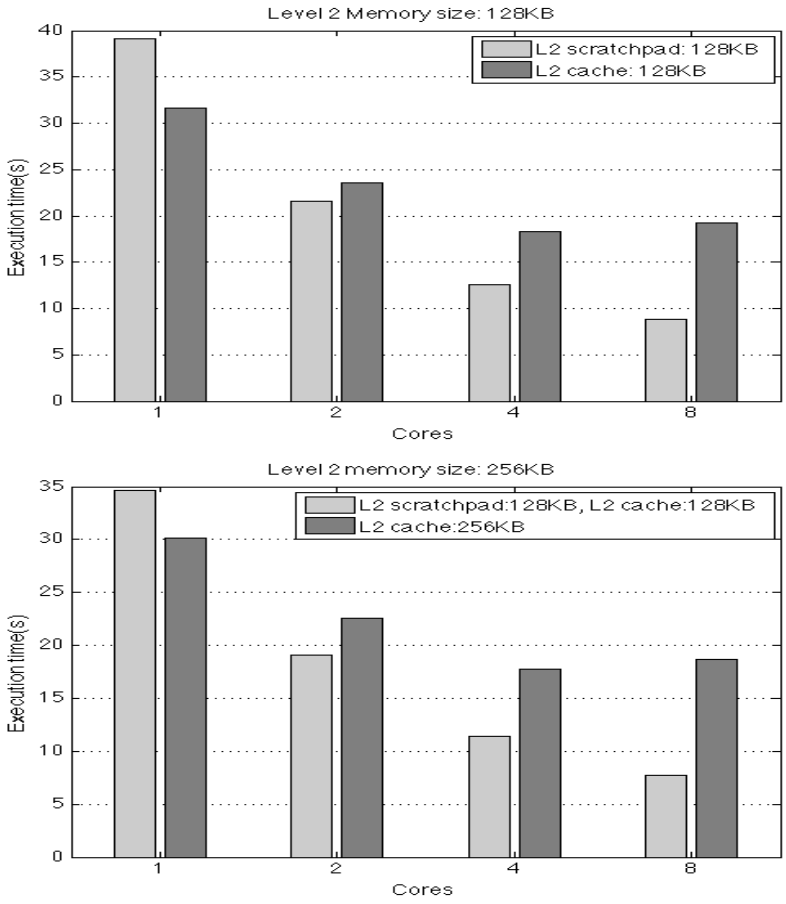


Figure 4.5: Comparing cache-based, scratchpad-based and hybrid configurations for FT.

## 4.5 Related Work

Other researchers have also explored the impact of different architectural parameters on the performance of applications in the NPB suite [76, 45, 10]. Wong et al. [76] focused on understanding how processor configuration affected the scaling of MPI

versions of the benchmarks on a cluster of Sun UltraSPARK nodes and on an SGI Origin 2000 system. They studied how the cache miss rate changed with cache size and machine size by collecting memory address traces from each application and feeding these into a cache simulator to simulate designed cache configurations. Their focus appeared to be on the level two cache, although it was not explicitly mentioned. They reported different levels of sensitivity to cache size for the applications considered. Kubota et al. [45] evaluated separate applications (MG and LU) from the suite and reported no significant effect from varying cache size.

In [10], the authors built a version of the benchmarks that used threads, and evaluated their performance on the Convex Exempler shared memory system. They measured the cache misses for increasing numbers of processors as well as the impact of cache line size on the performance. They observed that increasing cache line sizes could result in false sharing and produced more misses for the sample size. At the same time, for the larger problem size, there were more capacity misses for a given cache line size, but larger cache line sizes reduced the cache misses. A simulator was also used for their experiments.

The major difference between our work and previous studies is that the performance numbers that were reported in our study were results from execution on actual hardware, while previous studies were evaluated by simulation. While evaluations on simulators are the only option when hardware with the desired configurations are unavailable, such evaluations are prone to inaccuracy and not as reliable as careful measurement on a real system.

## 4.6 Summary

Performance characteristics of applications on large scale systems are often different from those on smaller systems. In this chapter, we reported the performance of a subset of the NAS parallel benchmarks while varying the memory configuration of the C6678 multicore DSP. We observed that while the level two cache size affected the performance in most cases, level one cache size had no significant impact on the execution time for the applications considered.

We observed performance improvement from utilizing scratchpad for FT, but this required an analysis of the source code in order to identify areas where scratchpad allocation will improve performance.

We were surprised at the scalability of the applications on the C6678, especially EP, considering its high degree of parallelism and limited data sharing. Investigating the factors that might have affected the performance of these applications forms part of our future work. In Section 4.4.4, we briefly touched on the issue of power in our comparison of the performance of the DSP versus the Opteron. A detailed analysis of the power consumption of the applications would provided even more insight to the benefits of configurable memory systems. This is another area that we plan to explore in future work.

# Chapter 5

# OpenMP-based Scratchpad Memory Management

## 5.1 Introduction

The previous chapter highlighted the performance benefits of using scratchpad memory. However, utilizing scratchpad memory for applications is more difficult, in comparison to caches, because the data and/or instructions that should be stored in scratchpad memory must be identified and explicitly specified in software. Static allocation strategies, where the contents of scratchpad memory are fixed throughout program execution, are usually not sufficient. Ideally, the contents of scratchpad memory should be allowed to change to accommodate situations where the memory requirements change across phases of the program, as well as cases where the entire memory items are too large to fit into scratchpad memory.

Dynamic approaches to utilizing scratchpad memory require that the memory items that are to be moved between main memory and scratchpad memory at a particular time be identified. In addition to this, other tasks that must be performed include explicit memory allocation in scratchpad memory to accommodate the data objects, copying (sections of) the data objects between main memory and scratchpad memory, updating the memory references in the program to reflect the new addresses of the data objects in scratchpad memory, and finally, deallocating the memory in scratchpad memory when the data objects no longer to reside there. All these tasks are entirely under software control and must be managed either by the programmer or a compiler. When left entirely to the programmer, accomplishing these tasks is both tedious and error-prone, which significantly reduces programmer productivity.

In this chapter, we present an OpenMP-based scratchpad management framework. The primary goal of this framework is to facilitate the use of scratchpad memory for array accesses within certain regions of OpenMP programs while requiring little to no additional information from the programmer. Our framework takes two approaches. First, we focus on code transformations that allow scratchpad memory to be automatically managed for arrays accessed within OpenMP loops. Second, we provide a high-level data region construct that allows programmers to express data movement in a high-level fashion with minimal impact on the structure of existing programs.

## 5.2 Automatic Scratchpad Memory Utilization

The OpenMP loop construct facilitates the scheduling of loop iterations to multiple threads. The specified loop schedule is used to divide the iteration space into *chunks* which get mapped to threads. While performing the computations included in their assigned chunks, threads may access private or shared variables, which are usually arrays. Each thread has access to only its own private copy of variables, but all threads have access to shared variables. The key idea of our approach is to identify portions of arrays that will be accessed during the execution of loop chunks, and map these portions to scratchpad memory. When we have shared and private scratchpad memory regions, we use the private scratchpad memory for distinct sections that are accessed during the execution of each thread's chunk of iterations. Shared scratchpad memory is reserved for sections that are accessed by more than one thread. This strategy prevents the replication of common data across multiple private scratchpad memory regions, which reduces the effective total size of scratchpad memory.

### 5.2.1 Detection of Array Access Patterns

The first step is to extract information about array accesses on a thread-by-thread basis. We focus on arrays with known dimensions and array accesses that are affine functions of loop indexes and non-loop dependent variables. Consider the parallel loop in Figure 5.1. Figure 5.2 shows the array sections accessed by each thread, assuming that we have a team of four threads. For each iteration of $i$, loop $j$ accesses array elements in the first dimension of array $a$ that are between $0$ and $n$. The

second dimension is accessed through loop $i$; since this is the loop associated with the OpenMP constructs, each thread will access only a portion of this dimension. For array $b$, each iteration of $i$ causes loop $j$ to access the $i^{th}$ element in the first dimension, with this access pattern replicated in the second dimension $n$ times. Therefore, each thread will access only a portion of elements in the first dimension, but will access all of the second dimension.

```
1 #pragma omp for
2 for (i = 0; i < n; i++)
3    for (j = 0; j < n; j++)
4          a[i][j] = 6 * b[j][i];
```

Figure 5.1: OpenMP example illustrating thread access patterns

We leverage our OpenMP translation for the loop construct (Section 3.2.2.2) to detect the array access patterns. To start with, we record each array reference, and whether it is a read or a write access. For parallel loops, the lower and upper bound values of loop iterators are generated either by the compiler (as in the case of static schedule with no chunk size specified) or the OpenMP runtime library. We record this, along with the lower and upper bound value for loops nested within the parallel loop. By substituting these loop bounds into the array reference, we derive the array section accessed by each thread. Returning to our example in Figure 5.1, we have reference a[i][j] and b[j][i]. Using the translation strategy discussed in Section 3.2.2.2, thread 0 gets a lower bound of *0* and an upper bound of *9* for $i$, and a lower bound

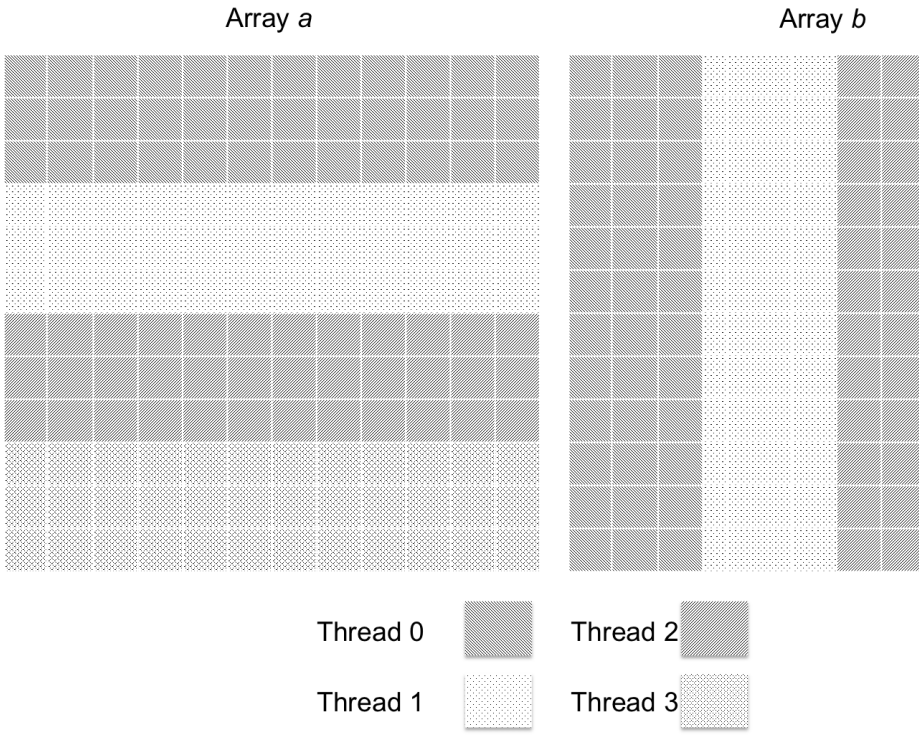


Figure 5.2: Sections of array accessed by each thread

of *0* and an upper bound of $n$ (=40) for $j$. Thread 0 therefore accesses 40 contiguous elements of $a$, starting from a[0][0], 10 times, and 10 contiguous elements of $b$, 40 times.

If these sections will fit in scratchpad, then statements to copy the sections are inserted into the code. Otherwise, we will attempt to keep only the section accessed in the $j$ loop in scratchpad memory.

We detect shared access by performing an intersection of the array sections. When threads access overlapping sections of memory, then the intersection of these array sections will be non-empty. Such array sections are candidates for a shared scratchpad memory region, if that region exists on the hardware platform.

### 5.2.2 Code Restructuring to Utilize Scratchpad Memory

At this point, we have the array sections that will be accessed in the loop (nest). The next step is generating memory allocation statements to allocate space in scratchpad. The data type and the size of the dimension is used to determine the size of each memory allocation. A corresponding de-allocation statement is also generated. The allocation and deallocation statements are placed before and after the loop to avoid unnecessary allocation/deallocation overheads.

Statements for copying data into or out of scratchpad memory are also generated depending on whether the access type within the loop nest is read or write. OpenMP's relaxed-consistency shared memory model means that a thread's temporary view of memory does not have to be consistent with memory at all times; memory synchronization is only achieved at specific synchronization points. The end of an OpenMP loop construct is one of such synchronization points. Therefore, portions of shared data can safely reside in private scratchpad memory until the next

synchronization point.

After the data allocation/copying statements have been inserted, the references within the innermost loop to locations that will be in scratchpad memory for the computation are modified to use the address in scratchpad memory instead of the address of the original array locations.

First, the bounds for the loops controlling the traversal of the data sections are adjusted to a zero offset. Then, each scratchpad memory reference is normalized to an offset of the base address obtained from the allocation. After these modifications, the loop iteration variables may be used to index the relevant arrays.

The resulting code generated by the compiler after the last two steps are applied is shown in Figure 5.3. Since the C6678 includes a Direct Memory Access subsystem, we use DMA transfers for copying data. A potential optimization is to overlap the data copying with actual computations by prefetching data that will be used in future iterations.

## 5.3 Memory Region Directive

The `memory region` directive is a new directive that provides the programmer with a high-level way to express data objects that should be in a memory region when executing an associated region of code. Memory regions are an abstraction of the distinct scratchpad memory regions within a memory hierarchy [15]. The compiler uses the directive to generate the required allocation and, if required, data movement

```
...
   for (temp_29 = temp_25; (temp_29 < temp_24); temp_29++)
   {
       edmaInitiateXfer(temp_31, ((*(omp_data_p27->a)) + temp_29 *
          (*(omp_data_p27->n))), ((unsigned)(*(omp_data_p27->n)) *
           8U), 1, 1, 1, 1, 1, 1, 0, 1);

       edmaInitiateXfer(temp_30, ((*(omp_data_p27->b)) + temp_29 *
          (*(omp_data_p27->n))), 8U, *(omp_data_p27->n), 1, ((
          unsigned)(*(omp_data_p27->n)) * 8U), 8U, 1, 1, 0, 1);

       edmaWait4Completion(0);
       for (temp_28 = 0; temp_28 < *(omp_data_p27->n); temp_28++)
       {
           temp_31[temp_28] += (double)(*(omp_data_p27->c)) * (
              temp_30[temp_28]);
       }

       edmaInitiateXfer((*(omp_data_p27->a) + temp_29 * (*(
          omp_data_p27->n))), temp_31, ((unsigned)(*(omp_data_p27
          ->n)) * 8U), 1, 1, 1, 1, 1, 1, 0, 1);

       edmaWait4Completion(0);
   }
...
```

Figure 5.3: Code generated from Figure 5.1 to use scratchpad

code. It also uses the directive to perform any necessary address updates to ensure correct access to the corresponding data objects in scratchpad. This approach is similar to the OpenMP approach of using directives to specify parallelism in applications. It removes the need for the programmer to deal with most of the *how* of scratchpad memory utilization, instead allowing him to focus mainly on *what* needs to be in scratchpad memory during the execution of specific program regions.

### 5.3.1 Syntax and Behavior of Memory Region Directive

A `memory region` directive is used to indicate a section of code that accesses data objects that will benefit from being allocated in (or moved to) a scratchpad memory region. The syntax of the `memory region` directive is as follows:

`#pragma memory region` *clause-list*

*structured-block*

where *clause-list* is at least one of the following:

`alloc(`*list*`)`

`in(`*list*`)`

`out(`*list*`)`

`inout(`*list*`)`

The `alloc` clause specifies only memory allocation within a scratchpad memory region for the listed data objects and does not imply any data movement into or out of the memory region. Data objects that are allocated in a scratchpad memory region can be initialized with their values before the memory region was encountered using the `in` clause. Similarly, the `out` clause indicates that values of data objects at the end of a memory region should be copied back to their location outside the memory region. The `inout` clause indicates that values of data objects be copied in at the start of the memory region, and copied back to their original location at the end of the memory region.

*list* is a comma-separated list of variables. For array variables, especially in cases where the entire array cannot fit in scratchpad, array sections may be specified in *list* using the Fortran-like array sections format:

*array*(*section-subscript-list*)

where *array* is the name of the array, and *section-subscript-list* is a list of one or more section subscripts indicating a set of elements along a particular dimension of the specified array. *section-subscript-list* is represented either as an array subscript, or with the following subscript triplet notation:

[*first-bound*] : [*last-bound*] [:stride]

Values for subscripts and subscript triplets must be scalar integer expressions. The stride is optional; when omitted it is assumed to be 1. This syntax provides for a simple yet flexible interface for expressing a variety of contents for scratchpad memory. Note that references to data objects outside memory regions are to the versions of the data objects in their original location. Regions outside of the bounds of the specified section should not be referenced within memory regions.

In Figure 5.4, we modify our matrix multiplication example to use of the `memory region` directive. For $A$ and $C$, elements 0 through $n$ in the $i$ dimension are copied to, and accessed via scratchpad memory. The same section of $C$ is copied back to main memory at the end of the memory region.

```
#pragma omp for private(i, j, k)
#pragma memory region inout(c(i:i,0:n)) in(a(i:i,0:n))
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
      {
         C[i][j] += A[i][k] * B[k][j];
      }
```

Figure 5.4: Matrix multiplication using memory region

Memory regions may be nested to cater to hierarchical scratchpad organizations. Figure 5.5 illustrates how a nested memory region maps to the scratchpad level that is immediately higher than the current scratchpad level. Each memory region in the code corresponds to the scratchpad level within the memory hierarchy with the same color.

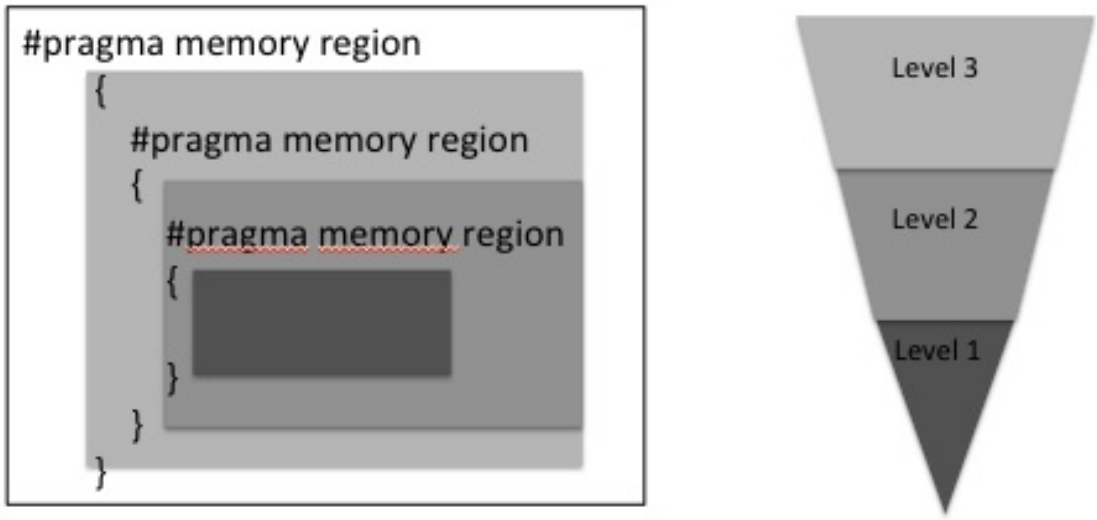


Figure 5.5: Nested memory regions

### 5.3.2 Compiler Implementation

The main translation tasks are (i) inserting calls to manage allocation and deallocation in the scratchpad memory region; (ii) inserting calls for copying data between memories; and (iii) updating the code within the memory region to reflect the correct addresses in scratchpad memory.

To tackle the first task, the size of memory that should be allocated will have to determined. If only a variable name is given, then the size corresponds to the size of the object in main memory. We currently support only statically allocated variables that expose the size of the variable to the compiler. If a section of data objects is specified, the compiler uses the subscript list to compute the size. Since we require that the subscripts are scalar integer expressions, we can express the size in the context of specified expressions. Using the size information, a data allocation function call for allocating a range of contiguous memory addresses is created and placed before the first statement of the associated memory region. A corresponding deallocation statement is placed after the last statement of the memory region.

Data objects that end up in scratchpad memory are organized in a contiguous manner, which may be different from their original form in main memory. We support both the standard *memcpy()* operation or DMA for achieving the copy operations. Depending on how the data section to be moved is specified, the compiler either generates a single *memcpy()* call to move contiguous data from the source address to the destination address, or it generates multiple *memcpy()* calls to move non-contiguous data.

We update the memory references within the memory region to reflect the addresses of the data objects's scratchpad locations instead of their addresses in main memory. This requires a traversal of the statements in the associated region, and checking whether there is a match between any encountered object references in the region and the objects specified in the directive. When there is a match, we rewrite the object reference as an offset to the address returned by the allocation function.

It is possible that memory allocation fails during execution. If this happens, the original version of the code is executed. Therefore, an if statement that checks the result of the allocation is used to determine whether to execute the original or the translated versions.

## 5.4 Evaluation

We compiled two versions of the matrix multiplication code in Figure 3.8—one with the automatic scratchpad utilization enabled, and the other with it disabled. When scratchpad utilization is enabled, rows of $C$ and $A$ and columns of $B$ are copied to scratchpad memory before the inner $k$ loop is executed. We compared the performance of both versions on the C6678. For both versions, level 1 cache size was set to 32KB, while we configured 256KB of the level 2 memory as either cache or scratchpad, depending on which version we were executing.

We show the performance of the matrix multiplication code for double-precision 256 by 256, 512 by 512, and 1024 by 1024 matrices in Figure 5.6. For all three problem sizes, we observed better performance with the versions that were compiled

to use scratchpad.

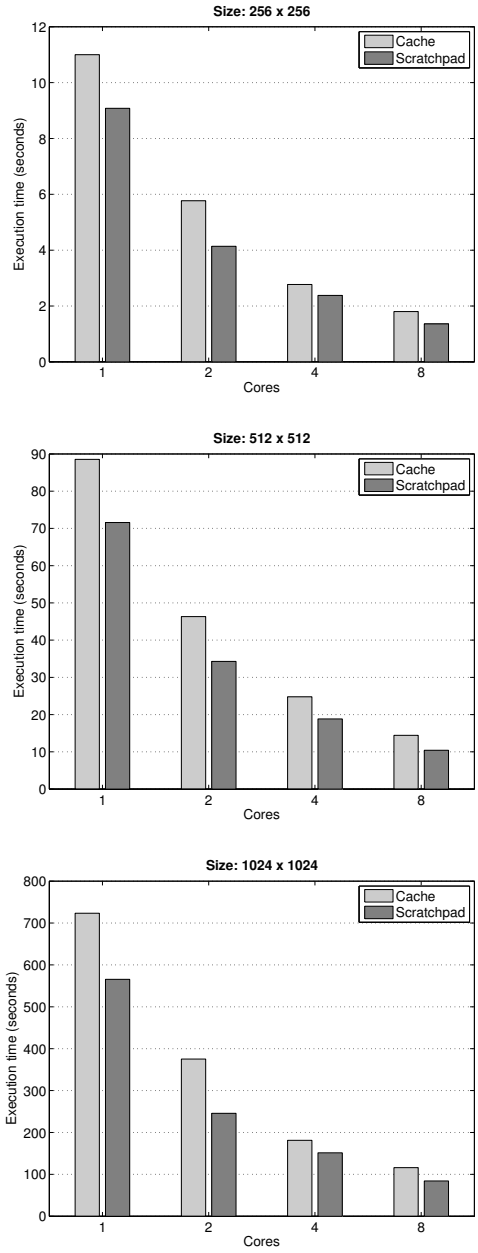


Figure 5.6: Performance of cache versus scratchpad version of matrix multiplication for various matrix sizes

We also used a five-point stencil algorithm for .

## 5.5 Limitations

Although our proposed framework can improve the performance of algorithms by utilizing scratchpad, the fundamental limitation is that it does not change the structure of the algorithm in order to exploit more data locality. This is not necessarily a limitation of the approach itself, but results from fact that our transformations are performed in the front-end of the compiler. At the front-end, we do not have access to control and data flow analyses information that will facilitate such transformations. However, an algorithm that already contains good locality of references can easily be adapted to our framework.

## 5.6 Related Work

A considerable amount of literature has been published on compiler support for efficient scratchpad memory allocation in embedded systems. Most of it has focused on allocation of data objects to scratchpad [63, 64, 42, 62, 74, 28, 50, 19], while some has focused on allocation of instructions [70, 13]. In general, the allocation schemes may be classified as static or dynamic. In the static scheme, the data layout in scratchpad remains fixed throughout program execution. All of these focused on scratchpad memory utilization in the context of a sequential programs only while our work focuses on the problem in the context of parallel OpenMP programs.

OpenMP-based implementations for processors with explicitly managed memory hierarchies have also been proposed. Marongiu and Benini [55] proposed a profile-based approach for allocating portions of large arrays to scratchpad memory after monitoring access patterns during an initial program execution. However, data allocation in their scheme is static and remains fixed throughout program execution. Their more recent work [56] proposed the use of high-level directives to partition arrays into distributed tiles and control the movement of array tiles across parallel regions. Liu and Chaudhary [54, 53] proposed OpenMP extensions that facilitated the use of memory transfer engines (MTEs) for DMA transfers in addition to extensions for expressing heterogeneity in the Software Scalable System on Chip (3SoC) from Cradle Technologies [25]. However, it is unclear how their proposed extensions could be used to manage the transfer of multiple data items. He et al. [35] implemented OpenMP for the MSC8156 multicore DSP from Freescale Semiconductors. In this work, they proposed additional data placement directives that control the placement of global variables, and data distribution directives that distribute arrays into core-local memory.

## 5.7 Summary

In this chapter, we presented a compiler framework for managing scratchpad memory regions within OpenMP programs. A technique for identifying appropriate array references that may be satisfied by scratchpad memory was proposed along with a high-level directive for specifying (sections of) objects that should be in scratchpad

memory. Our experimental results show that our compiler-generated code that utilizes scratchpad memory achieves significant performance improvement compared to the cache based versions of the same code. For future work, we plan to extend the robustness of our implementation by integrating it into later phases in the compilation tool chain where more sophisticated analyses can be leveraged.

# Chapter 6

# Conclusion

With the increasing adoption of multicore processors in the embedded domain, the need for high-level programming models that assist programmers with parallelization and memory management are more important. Programmers should be able to focus on writing their algorithms without dealing with all of the low-level aspects of coordinating parallelism and managing memory.

In this dissertation, we developed a compiler implementation of OpenMP, a high-level programming model, that is suitable for for embedded multicore processors. This fully functional implementation is included in the C6000 compiler that currently benefits a large community of embedded systems developers.

We extended this implementation with a framework that facilitates utilizing scratchpad memory while requiring little or no extra information from the programmer. Our experimental results show improved performance with our framework over

cache-based executions.

We also analyzed the effect of various memory configurations on the performance of applications from a standard scientific benchmark.

## 6.1 Future Work

We are looking forward to extending this work in the following main areas.

**Incorporating Data Locality with Scratchpad Memory Utilization** In terms of utilizing scratchpad memory, while our experimental results are encouraging, the fact that we do not perform any optimizations to improve data reuse significantly limits the performance of the applications. On possibility is incorporating our framework into loop analyses frameworks that determine optimal tile sizes for hierarchical architectures. Another area that we plan to explore is data reuse between threads. Consider an example where all threads access an array that is too large to fit into scratchpad memory. The data access patterns of the threads could be modified such that all/most threads access the same section during a time frame rather than unique sections.

**Adaptive Cache/Scratchpad Reconfiguration** There are benefits to both cache and scratchpad configurations and applications can simultaneously benefit from both configurations. When it is possible to have both configurations, an important question becomes how much memory to assign to each configuration in order to get

good performance. For future work, we will explore solutions to dynamically adapt the size of cache and scratchpad memory based on the requirements of (phases of) applications.

# Bibliography


[1] Bison. http://www.gnu.org/software/bison/. Accessed : 6 July 2013.

[2] Edison Design Group. http://www.edg.com. Acccessed: 30 November 2012.

[3] IBM C and C++ Compilers Family. http://www-03.ibm.com/software/products/us/en/ccompfami. Accessed : 6 July 2013.

[4] IBM Fortran Compilers Family. http://www-03.ibm.com/software/products/us/en/fortcompfami. Accessed : 6 July 2013.

[5] Intel Compilers. http://software.intel.com/en-us/intel-compilers. Accessed : 6 July 2013.

[6] OpenMP Architecture Review Board. http://openmp.org/wp/. Acccessed: 1 September 2012.

[7] Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html. Accessed : 6 July 2013.

[8] Pathscale EKOPath Compiler Suite. http://www.pathscale.com/ekopath-compiler-suite. Accessed : 6 July 2013.

[9] The GNU Compiler Collection. http://gcc.gnu.org/. Accessed : 6 July 2013.

[10] G. A. Abandah. *Characterizing Shared-memory Applications: A Case Study of NAS Parallel Benchmarks*. Hewlett Packard Laboratories, 1997.

[11] S. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12): 66 –76, December 1996.

[12] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The Performance Implications of Thread Management Alternatives for Shared-memory Multiprocessors. *IEEE Transactions on Computers*, 38(12): 1631–1644, Dec. 1989.

[13] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A Post-Compiler Approach to Scratchpad Mapping of Code. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 259–267. ACM, 2004.

[14] A. Aribuki, S. Chandrasekaran, and B. Chapman. Improving Heterogeneous Multicore Programming using OpenMP. In *Proceedings of SRC's Techcon 2011*, June 2011. `http://www.src.org/library/publication/p060398/`.

[15] A. Aribuki, E. Stotzer, and E. Leiss. High-Level Directives for Efficiently Utilizing Scratchpad Memory. In *50th ACM/EDAC/IEEE Design Automation Conference Work-in-Progress*, June 2013.

[16] ARM. The ARM Cortex-A9 Processors. White Paper, September 2009.

[17] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos Mercurium: A Research Compiler for OpenMP. In *Proceedings of the European Workshop on OpenMP (EWOMP '04)*, Vol. 8, Stockholm, Sweden, October 2004.

[18] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems. In *Proceedings of the tenth international symposium on Hardware/-software codesign*, CODES '02, 73–78, New York, NY, USA, 2002. ACM.

[19] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1–10. ACM, 2008.

[20] R. Blumofe and C. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of 35th Annual Symposium on Foundations of Computer Science*, 356 –368, November 1994.

[21] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, 207–216, New York, NY, USA, 1995. ACM.

[22] C. Brunschen and M. Brorsson. OdinMP/CCp - A Portable Implementation of OpenMP for C. *Concurrency - Practice and Experience*, 12(12): 1193–1203, 2000.

[23] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer. Implementing OpenMP on a High Performance Embedded Multicore MPSoC. In *IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009*, 1–8. IEEE, 2009.

[24] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, 57–66. IEEE, 2008.

[25] Cradle Technologies Inc. The Software Scalable System on Chip (3SoC) Architecture. White Paper, March 2002.

[26] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1): 46–55, January-March 1998.

[27] J. Diamond, M. Burtscher, J. McCalpin, B.-D. Kim, S. Keckler, and J. Browne. Evaluation and Optimization of Multicore Performance Bottlenecks in Supercomputing Applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 32 –43, April 2011.

[28] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An Integrated Hardware/Software Approach for Run-Time Scratchpad Management. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, 238–243, New York, NY, USA, 2004. ACM.

[29] Freescale Semiconductor. MSC8156. Data Sheet, December 2011.

[30] Freescale Semiconductor. P2020. Data Sheet, March 2012.

[31] M. Frigo, C. Leiserson, and K. Randall. The Implementation of the Cilk-5 Multithreaded Language. *ACM Sigplan Notices*, 33(5): 212–223, 1998.

[32] A. Funk, V. Basili, L. Hochstein, and J. Kepner. Application of a Development Time Productivity Metric to Parallel Software Development. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, 8–12, New York, NY, USA, 2005. ACM.

[33] D. Geer. Chip Makers Turn to Multicore Processors. *Computer*, 38(5): 11 – 13, May 2005.

[34] P. F. Gorder. Multicore Processors for Science and Engineering. *Computing in Science and Engineering*, 9: 3–7, 2007.

[35] J. He, W. Chen, G. Chen, W. Zheng, Z. Tang, and H. Ye. OpenMDSP: Extending OpenMP to Program Multi-Core DSP. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 288 –297, October 2011.

[36] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. Hollingsworth, and M. Zelkowitz. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *Proceedings of the ACM/IEEE Conference on Supercomputing, 2005*, 35–35, 2005.

[37] IEEE Portable Applications Standards Committee and others. IEEE Std 1003.1 c-1995, Threads Extensions, 1995.

[38] F. D. Igual, M. Ali, A. Friedmann, E. Stotzer, T. Wentz, and R. A. van de Geijn. Unleashing the High-performance and Low-power of Multi-core DSPs for General-purpose HPC. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2012*, 1–11, November 2012.

[39] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical Report NAS-99-011, NASA Advanced Supercomputer (NAS) System Division NASA Ames Research Center, October 1999.

[40] S. C. Johnson. *Yacc: Yet Another Compiler-Compiler*, Vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[41] S. C. Johnson. Yacc: Yet Another Compiler Compiler. *Unix Programmer's Manual*, 2: 353–387, 1979.

[42] M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-pad Memory Space. In *Design Automation Conference, 2001. Proceedings*, 690–695. IEEE, 2001.

[43] W. Kim and M. Voss. Multicore Desktop Programming with Intel Threading Building Blocks. *Software, IEEE*, 28(1): 23 –31, January-February 2011.

[44] C. Kruskal, L. Rudolph, and M. Snir. Efficient Synchronization of Multiprocessors with Shared Memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(4): 579–601, 1988.

[45] K. Kubota, K. Itakura, M. Sato, and T. Boku. Practical Simulation of Large-Scale Parallel Programs and its Performance Analysis of the NAS Parallel

Benchmarks. In D. Pritchard and J. Reeve, editors, *Euro-Par'98 Parallel Processing*, Vol. 1470 of *Lecture Notes in Computer Science*, 244–254. Springer Berlin Heidelberg, 1998.

[46] J. LaGrone, A. Aribuki, C. Addison, and B. M. Chapman. A Runtime Implementation of OpenMP Tasks. In *The International Workshop on OpenMP (IWOMP)*, 165–178, 2011.

[47] J. LaGrone, A. Aribuki, and B. Chapman. A Set of Microbenchmarks for Measuring OpenMP Task Overheads. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Vol. II, 594–600, July 2011.

[48] E. A. Lee. The Problem with Threads. *Computer*, 39(5): 33–42, May 2006.

[49] C. Leiserson. The Cilk++ Concurrency Platform. *The Journal of Supercomputing*, 51(3): 244–257, 2010.

[50] L. Li, H. Feng, and J. Xue. Compiler-Directed Scratchpad Memory Management via Graph Coloring. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(3): 9, 2009.

[51] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An Optimizing, Portable OpenMP Compiler. *Concurrency and Computation: Practice and Experience*, 19(18): 2317–2332, 2007.

[52] C. Liao, D. Quinlan, T. Panas, and B. Supinski. A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. In M. Sato, T. Hanawa, M. Müller, B. Chapman, and B. Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, Vol. 6132 of *Lecture Notes in Computer Science*, 15–28. Springer Berlin Heidelberg, 2010.

[53] F. Liu and V. Chaudhary. A Practical OpenMP Compiler for System on Chips. In *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming*, WOMPAT'03, 54–68, Berlin, Heidelberg, 2003. Springer-Verlag.

[54] F. Liu and V. Chaudhary. Extending OpenMP for Heterogeneous Chip Multiprocessors. In *ICPP'03*, 161–161, 2003.

[55] A. Marongiu and L. Benini. Efficient OpenMP Support and Extensions for MPSoCs with Explicitly Managed Memory Hierarchy. In *Proceedings of Design, Automation and Test in Europe*, 809–814, 2009.

[56] A. Marongiu and L. Benini. An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs. *IEEE Transactions on Computers*, 61(2): 222–236, 2012.

[57] G. Narlikar and G. Blelloch. Pthreads for Dynamic and Irregular Parallelism. In *IEEE/ACM Conference on Supercomputing,1998 (SC98)*, 31, November 1998.

[58] D. Novillo. OpenMP and Automatic Parallelization in GCC. In *GCC developers summit*, 2006.

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

[60] K. Olukotun and L. Hammond. The Future of Microprocessors. *Queue*, 3(7): 26–29, Sept. 2005.

[61] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.1, July 2011.

[62] P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. Kjeldsberg. Data and Memory Optimization Techniques for Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2): 149–206, 2001.

[63] P. Panda, N. Dutt, and A. Nicolau. Efficient Utilization of Scratch-pad Memory in Embedded Processor Applications. In *Proceedings of the 1997 European Conference on Design and Test*, 7. IEEE Computer Society, 1997.

[64] P. Panda, N. Dutt, and A. Nicolau. On-chip vs. Off-chip Memory: The Data Partitioning Problem in Embedded Processor-based Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3): 682–704, 2000.

[65] P. R. Panda, A. Nicolau, and N. Dutt. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[66] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Designower electronics and design*, ISLPED '00, 90–95, New York, NY, USA, 2000. ACM.

[67] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, Incorporated, 2007.

[68] A. Robison, M. Voss, and A. Kukanov. Optimization via Reflection on Work Stealing in TBB. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 1 –8, April 2008.

[69] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proceedings of the 1st European Workshop on OpenMP*, 32–39, 1999.

[70] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *15th International Symposium on System Synthesis*, 213–218. IEEE, 2002.

[71] Texas Instruments Inc. TMS320C6000 Optimizing Compiler v7.4 User's Guide, July 2012.

[72] Texas Instruments Inc. TMS320C6678. Data Manual, February 2012.

[73] X. Tian, M. Girkar, S. Shah, D. Armstrong, E. Su, and P. Petersen. Compiler and Runtime Support for Running OpenMP Programs on Pentium- and Itanium-Architectures. In *Proceedings of International Parallel and Distributed Processing Symposium, 2003*, 9 pp., 2003.

[74] S. Udayakumaran and R. Barua. Compiler-Decided Dynamic Memory Allocation for Scratch-pad Based Embedded Systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 276–286. ACM, 2003.

[75] J. von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4): 27–75, 1993.

[76] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, Supercomputing '99, New York, NY, USA, 1999. ACM.

[77] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Archititecture News News*, 23(1): 20 – 24, Mar. 1995.

[78] S.-H. Yang, M. Powell, B. Falsafi, and T. N. Vijaykumar. Exploiting Choice in Resizable Cache Design to Optimize Deep-submicron Processor Energy-delay. In *Proceedings of Eighth International Symposium on High-Performance Computer Architecture, 2002*, 151–161, 2002.

[79] C. Zhang, F. Vahid, and W. Najjar. A Highly Configurable Cache Architecture for Embedded Systems. In *Proceedings of 30th Annual International Symposium on Computer Architecture, 2003*, 136–146, 2003.