He JZ, Chen WG, Chen GR *et al.* OpenMDSP: Extending OpenMP to program multi-core DSPs. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 29(2): 316–331 Mar. 2014. DOI 10.1007/s11390-014-1433-x

OpenMDSP: Extending OpenMP to Program Multi-Core DSPs

Jiang-Zhou He[1] (何江舟), *Student Member*, *CCF*, Wen-Guang Chen[1] (陈文光), *Member*, *CCF*, *ACM*, *IEEE*
Guang-Ri Chen[2] (陈光日), Wei-Min Zheng[1] (郑纬民), *Fellow*, *CCF*, *Member*, *ACM*, *IEEE*
Zhi-Zhong Tang[1] (汤志忠), *Member*, *CCF*, and Han-Dong Ye[2] (叶寒栋)

[1] *Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China*

[2] *Huawei Technologies Co. Ltd., Shenzhen 518129, China*

E-mail: hejz07@mails.tsinghua.edu.cn; cwg@tsinghua.edu.cn; chenguangri@huawei.com;
{zwm-dcs, tzz-dcs}@tsinghua.edu.cn; hye@huawei.com

Received March 26, 2013; revised November 29, 2013.

Abstract Multi-core digital signal processors (DSPs) are widely used in wireless telecommunication, core network transcoding, industrial control, and audio/video processing technologies, among others. In comparison with general-purpose multi-processors, multi-core DSPs normally have a more complex memory hierarchy, such as on-chip core-local memory and non-cache-coherent shared memory. As a result, efficient multi-core DSP applications are very difficult to write. The current approach used to program multi-core DSPs is based on proprietary vendor software development kits (SDKs), which only provide low-level, non-portable primitives. While it is acceptable to write coarse-grained task-level parallel code with these SDKs, writing fine-grained data parallel code with SDKs is a very tedious and error-prone approach. We believe that it is desirable to possess a high-level and portable parallel programming model for multi-core DSPs. In this paper, we propose OpenMDSP, an extension of OpenMP designed for multi-core DSPs. The goal of OpenMDSP is to fill the gap between the OpenMP memory model and the memory hierarchy of multi-core DSPs. We propose three classes of directives in OpenMDSP, including 1) data placement directives that allow programmers to control the placement of global variables conveniently, 2) distributed array directives that divide a whole array into sections and promote the sections into core-local memory to improve performance, and 3) stream access directives that promote big arrays into core-local memory section by section during parallel loop processing while hiding the latency of data movement by the direct memory access (DMA) of a DSP. We implement the compiler and runtime system for OpenMDSP on FreeScale MSC8156. The benchmarking results show that seven of nine benchmarks achieve a speedup of more than a factor of 5 when using six threads.

Keywords OpenMP, multi-core digital signal processor, data parallelism, Long Term Evolution

1 Introduction

1.1 Importance of Data Parallelism for Multi-Core DSPs

Multi-core digital signal processors (DSPs) are widely used in wireless telecommunication, core network transcoding, industrial control, and audio/video processing technologies, among others. In comparison with general-purpose multi-processors, multi-core DSPs usually have a more complex memory hierarchy, such as on-chip core-local memory and non-cache-coherent shared memory[1]. On-chip core-local memory can usually be addressed with different address spaces and cannot be accessed directly by other cores. Non-cache-coherent shared memory also significantly enhances the complexity of memory management for shared data access since programmers are required to maintain the coherence of data manually. As a result, it is very challenging to write efficient multi-core DSP applications.

The state-of-the-art approach used to program multi-core DSPs is based on proprietary vendor software development kits (SDKs), which only provide low-level, non-portable primitives. It is common practice to use these SDKs to provide coarse-grained task-level parallelism for applications. For example, the next generation of wireless telecommunication protocol, Long Term Evolution (LTE)[2], is a very important application of multi-core DSPs. Currently, developers parallelize LTE base station applications at the task-level. Fig.1 is a partial task graph of an LTE base station physical layer uplink. The input of each fast Fourier transform (FFT) task is the carrier waveform received from each antenna port. The waveforms of different carriers are fed into FFT tasks one after another. Deve-

Regular Paper

This work was supported by the National High Technology Research and Development 863 Program of China under Grant No. 2012AA010901 and the National Natural Science Foundation of China under Grant No. 61103021.

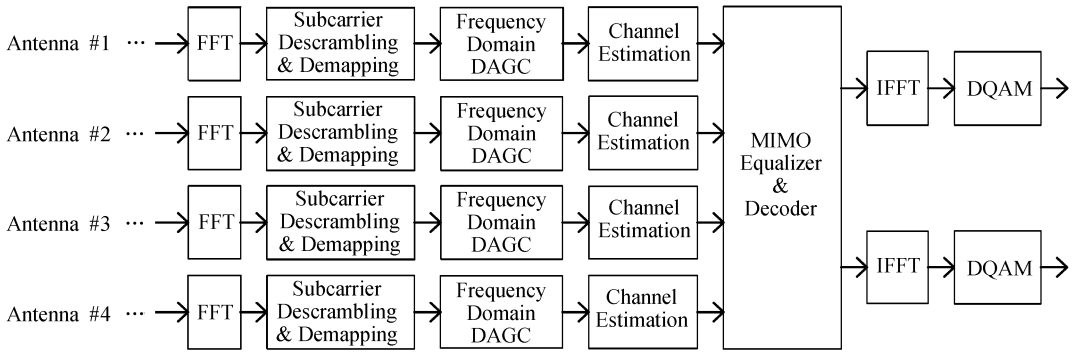


Fig.1. Partial task graph of an LTE base station physical layer uplink.

lopers map the tasks for a waveform from different antennas to different cores to exploit task parallelism, or map the tasks of different stages to different cores to exploit pipeline parallelism between adjacent carriers, or both.

Although task parallelism and pipeline parallelism appear to work well, both are insufficient to leverage fine-grain parallelism inside tasks. The emergence of low-power multi-core DSPs demands programmers to exploit data parallelism as well. For example, the PicoChip PC205[1] has 248 cores and works at a frequency of 280MHz. While the low frequency gives the chip a big power efficiency boost over traditional multi-core DSPs, it also imposes additional challenges to programmers. Without employing data parallelism in applications, it is very difficult for this chip to meet the latency requirement of LTE signal processing. Thus, we believe it is critical to support data parallelism in addition to task parallelism and pipeline parallelism for future multi-core DSP platforms.

1.2 Problems of Currently Available Programming Models for Multi-Core DSPs

In modern industry, the common practice of programming multi-core DSPs is based on the use of low-level, proprietary vendor SDKs. While such SDKs provide reasonable abstraction for task parallelism and pipeline parallelism, they are usually too low-level when used for the expression of data parallelism. An example of data parallel code written with the SDK of FreeScale MSC8156 is shown in Fig.2.

From the example, it can be seen that programmers must manually write the code used for barrier, loop dividing and scheduling, data reduction, data sharing and synchronization, making programming very tedious and error-prone. The other issue is portability. The language extension defined in these proprietary SDKs is not portable to the DSPs of other vendors.

Researchers have proposed various programming models to solve these problems. SoC-C[3] is a programming model that was developed for heterogeneous multi-core systems. It has support for task parallelism and pipeline parallelism but lacks support for data parallelism. StreamIt[4] is another influential programming model used for stream applications. It defines an elegant dataflow programming model for the exploitation of data parallelism, task parallelism and pipeline parallelism. However, the problems with StreamIt are twofold: 1) It is difficult to incrementally change existing code to StreamIt. As a result, a significant portion of the legacy code requires rewriting. 2) StreamIt is based on the static data-flow model. In DSP applications, there are scenarios that require a dynamic level of parallelism and task binding.

```
 1 __attribute__((section(".m3_shared"))) int a[1024];
 2 __attribute__((section(".m3_shared"))) int sum = 0;
 3 __attribute__((section(".m3_shared"))) spinlock l1;
 4 void barrier(int count) { ... }
 5 void sum_a() {
 6   int thread_id = get_core_id() - 2;
 7   int num_threads = 4;
 8   int local_sum = 0;
 9   int lower = 1024 * thread_id / num_threads;
10   int upper = 1024 * (thread_id + 1) / num_threads;
11   if (thread_id == 0) sum = 0;
12   barrier(num_threads);
13   for (i = lower; i < upper; i++)
14     local_sum += a[i];
15   acquire_spinlock(&l1);
16   sum += local_sum;
17   release_spinlock(&l1);
18   barrier(num_threads);
19 }
```

Fig.2. Data parallelized code to add up an array, based on the SDK of FreeScale MSC8156. `sum_a()` is intended to run on core #2 to core #5. The contents of `barrier()` are omitted.

1.3 Extend OpenMP to Support Multi-Core DSP Programming

The purpose of this research is to provide a parallel programming model for multi-core DSPs that has the

following features: 1) It must allow incremental change to support data parallelism on existing task/pipeline parallel code written with SDKs. 2) High level abstractions must be available to avoid tedious loop bound calculation and synchronization for data parallel code. 3) It must be portable among different DSP platforms.

We propose to extend OpenMP to support multi-core DSP programming. OpenMP[①] is a widely adopted industrial standard. Extension based on OpenMP provides a good chance for portability. OpenMP is powerful for the expression of data parallelism. Additionally, OpenMP provides high-level abstractions, which can prevent programmers from tedious work, such as calculating parallel loop bounds for each thread. With OpenMP, programmers can add a few annotations to sequential code for incremental parallelization, which matches our goal of incremental parallelization very well.

Standard OpenMP only supports cache-coherent shared memory systems. In multi-core DSPs, core-local memory and non-cache coherent shared memory both exist, imposing two key challenges to OpenMP on multi-core DSPs.

• *Core-Local Memory.* On general-purpose multi-core CPU systems, shared variables with automatic storage duration residing in the stack of master thread can easily be accessed by other worker threads because of a unified address space. However, on most multi-core DSPs, the core-local memory has different memory spaces with different memory hierarchies. The stack resides on core-local memory and cannot be accessed from other cores. If we must place shared variables with automatic storage duration in the stack, we require a way to make the stack accessible to other threads.

• *Non-Cache Coherent Shared Memory.* On general-purpose multi-core CPU systems, the shared memory is cache coherent which can boost the speed of shared data accessing when the data are not really shared during a given period of time; accordingly, the accesses will be cache hit without accessing the main memory. However, the shared memory of multi-core DSPs is not cache coherent; as a result, each access to the variables in shared memory should be a slow shared memory access if there is no opportunity to place some data within the fast core-local memory. We must address this issue in the design of our extension.

In this paper, we present OpenMDSP, an extension of OpenMP 2.5 for multi-core DSPs. We also have implemented an OpenMDSP compiler and runtime system, OMDPFS, for FreeScale MSC8156, which is a DSP processor with six cores. Our paper provides three main contributions:

1) We develop an approach for standard OpenMP 2.5 programs to run correctly on multi-core DSPs without modification. We design and implement compiler transformation and runtime mechanism so that all standard OpenMP 2.5 directives and APIs retain the same semantics on multi-core DSPs. This work allows us to annotate the current sequential code of different tasks to support data parallelism.

2) To improve the performance of the OpenMP code, we propose OpenMDSP, which is an extension of OpenMP 2.5, to allow for the optimized usage of the complex memory hierarchy of multi-core DSP systems. In particular, we support core-local memory and non-cache coherent shared memory with three new classes of directives of OpenMDSP, including:

• data placement directives that allow programmers to control conveniently the placement of global variables;

• distributed array directives that divide the whole array into sections and promote the sections into core-local memory to improve performance, and

• stream access directives that promote a big array into core-local memory section by section during parallel loop processing.

3) We implement the compiler and runtime system for OpenMDSP on FreeScale MSC8156 efficiently. Nine benchmarks are used to evaluate the performance of OpenMDSP. Seven out of nine benchmarks achieve a speedup of more than a factor of 5 with 6 threads.

The rest of the paper is organized as follows. Section 2 defines the OpenMDSP language extensions. Section 3 states the design and implementation of OMDPFS. We evaluate its implementation in Section 4, and we discuss the limitations in Section 5. In Section 6, we review related work, and Section 7 concludes the paper.

2 OpenMDSP Language Extensions

2.1 Overview of OpenMDSP Extensions

OpenMDSP is designed based on OpenMP 2.5 and inherits the execution model, memory model, directives and API functions defined in the OpenMP 2.5 specification. As stated in Section 1, to expose the memory hierarchy for programmers, we have extended OpenMP by a few new directives as shown in Fig.3.

Data placement directives (*alloc-directive* and *defaultalloc-directive*) provide a unified way for programmers to control the placement of certain variables. Besides placing variables on a specified memory hierarchy during their full life cycle, another common pattern of DSP applications is the temporary loading of shared variables into core-local memory for faster processing.

[①]The OpenMP API specification for parallel programming. http://www.spec.org, Nov. 2013.

alloc-directive ::=
  #pragma domp alloc(*place*, *var-list*) ***NL***
place ::= **corelocal** | **chipshare** | **offshare**
var-list ::= ***variable*** | ***variable***, *var-list*
defaultalloc-directive ::=
  #pragma domp defaultalloc *defalloc-placedef*$^+$ ***NL***
defalloc-placedef ::= **normal**(*place*) | **threadprivate**(*place*)
distribute-directive ::=
  #pragma domp distribute(*var-list*) *dist-clause** ***NL***
    statement
dist-clause ::= **size**(***expression***) | **peek**(***expression***, ***expression***)
  | **copyin** | **copyout** | **bulk**(***expression***) | **nowait**
for-respect-directive ::=
  #pragma domp for respect(*variable*) *forres-clause** ***NL***
    for-statement
forres-clause ::= **private**(*var-list*) | **firstprivate**(*var-list*)
  | **lastprivate**(*var-list*) | **reduction**(***operator***:*var-list*)
  | **nowait**
omp-parallel-for-directive [REDEFINE] =
  #pragma omp parallel for ***omp-parallel-for-clause**** ***NL***
  stream-directive*
    for-statement
omp-for-directive [REDEFINE] =
  #pragma omp for ***omp-for-clause**** ***NL***
  stream-directive*
    for-statement
stream-directive ::=
  #pragma domp stream(*var-list*) *stream-clause** ***NL***
stream-clause ::= **size**(***expression***) | **rate**(***expression***)
  | **peek**(***expression***, ***expression***) | **copyin** | **copyout**
  | **nowait**

Fig.3. OpenMDSP syntax extensions. *omp-parallel-for-directive* and *omp-for-directive* are redefinitions for `omp for` directive defined in the OpenMP specification, while others are definitions for new directives.

As shown in Table 1, several features of OpenMDSP allow programmers to easily cache shared data by core-local memory. Some features aim to load the entire shared dataset into core-local memory within one step, while others load data window by window, during the processing of the cached data, to deal with the situations in which the specified data is too large to fit into the core-local memory entirely. Meanwhile, the features also can be categorized by whether they divide the specified data for different cores. Therefore, four kinds of language features are available in OpenMDSP for caching shared data by core-local memory. The `private`, `firstprivate`, `lastprivate` and `copyprivate` clauses already defined in the OpenMP specification create a private copy of the shared data. In OpenMDSP, thread-private data are held in the core-local memory by default; as a result, these clauses naturally cache the shared data by core-local memory. The `distribute` directive and `stream` directive, which will be covered in Subsection 2.3 and Subsection 2.4 respectively, feature data caching ability in other manners.

Table 1. OpenMDSP Features Used to Load Shared Variables into Core-Local Temporarily

How to Access Data	How to Cache Data	
	Cache Entire Data	Cache Data Window by Window
Access data by one core	`private`, `firstprivate`, `lastprivate`, `copyprivate`	`stream` with sequential loop
Access data by different cores	`distribute`	`stream` with parallel loop

2.2 Extension for Data Placement

With OpenMP directives, programmers cannot specify mapping between variables and memory levels. In OpenMDSP, by default, shared variables with a static storage duration are placed in on-chip shared memory, while thread-private variables are placed in core-local memory. However, sometimes programmers must change the default placement. For example, the core-local memory may lack sufficient space to carry all thread-private variables; as a result, some of the thread-private variables must reside in either the on-chip shared memory or the off-chip shared memory.

We introduce the `alloc` directive for data placement. *place* can be `corelocal`, `chipshare` or `offshare` to indicate core-local memory, on-chip shared memory or off-chip shared memory, respectively.

We introduce another directive, named `defaultalloc`, to specify the default data placement location. The `defaultalloc` directive takes effect for all variables with a static storage duration, defined after that directive until the next `defaultalloc` directive, which overrides the setting.

2.3 Extension for Distributed Array

Shared data is stored in either the on-chip shared memory or the off-chip shared memory, which has a significant latency. We have observed that in a class of parallel algorithms, such as matrix operations, each thread only need to access a portion of an array. We have defined the `distribute` directive for such situation.

The `distribute` directive is used to create distributed duplications for a shared array in the core-local memory during the execution of its following statement. Any access to such an array in the following statement is performed by private duplication, which improves the performance. As shown in Fig.4, the whole array is divided into n contiguous sections, one for each thread,

where n is the number of threads. For each thread, we call the corresponding section the *main section* for this thread. In the statement followed by the `distribute` directive each thread is only allowed to access its main section of an array unless a `peek` clause is present.

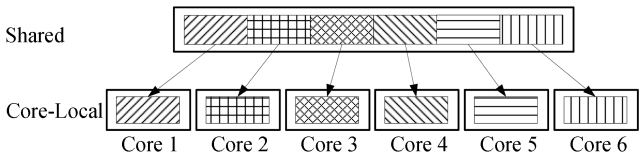


Fig.4. `distribute` directive creates distributed duplications for a shared array in the core-local memory.

Each variable placed in the `distribute` clause is the name of a shared array or a pointer to the first element of a shared array. If a pointer is placed here, the `size` clause is required to specify the size of the array to which it points. `size` clause is not needed for a static array.

The `peek` clause allows one thread to read some elements before and after the main section. The two expressions should specify the number of elements allowed that are to be read before and after. This clause is designed for algorithms in which the access ranges for an array in adjacent iterations intersect.

The `copyin` clause copies the data in the shared data into private memory before the execution of *statement*. Without `copyin`, the initial value of the private duplication is undefined. The `copyout` clause copies the data back to the shared memory; the original values in the shared memory do not change when `copyout` is not specified. `copyout` only takes effect for the main section, while `copyin` also takes effect based on the amount of elements as specified in `peek` clause.

The `bulk` clause is used to specify the minimal grain for the array range division. The size of each main section is guaranteed to be a multiple of the value of the specified expression except the last section. The default minimal grain is 1.

Like the `parallel`, `for` and `single` directives defined in OpenMP, `nowait` is used to remove the implicit barrier after execution of *statement*.

OpenMDSP provides another directive, `for respect`, which allows programmers to write code to iterate conveniently through the main section of a distributed array. The effect of the `domp for respect` directive is similar to the `omp for` directive, which parallelizes the following *for-statement*. Like `omp for` directive, `private`, `firstprivate`, `lastprivate`, `reduction` and `nowait` clauses are applicable. The distinct feature of `domp for` is that it divides the loop range consistent with the distribution of the array specified inside the braces after "`respect`". More precisely, the loop range in each thread is guaranteed to be the intersection of the main section of this thread for the specified array and the whole loop range. Fig.5 presents an example of the `distribute` and `for respect` directives.

```
 1 double a[M][N];
 2 double b[N][P];
 3 double c[M][P];
 4 void compute_c() {
 5   int i, j, k;
 6   double s;
 7 #pragma omp parallel firstprivate(b)
 8 #pragma domp distribute(a) copyin
 9 #pragma domp distribute(c) copyout
10 #pragma domp for respect(a)
11   for (i = 0; i < M; i++) {
12     for (j = 0; j < P; j++) {
13       s = 0;
14       for (k = 0; k < N; k++)
15         s += a[i][k] * b[k][j];
16       c[i][j] = s;
17     }
18   }
19 }
```

Fig.5. Matrix multiplication, an example of `distribute` and `for respect`.

2.4 Extension for Stream Access

Although the extension of the distributed array is helpful for reducing the access latency for shared array, it cannot work if the main section is too large to reside in the core-local memory. However, we observed that for access patterns applicable to `distribute` and `for respect`, large shared arrays also can be fetched in the core-local memory or written back piecewise as needed. This is the purpose of the `stream` directive.

The `stream` directive can either appear before any sequential `for` statement or can live together with the `omp parallel for` or `omp for` directive. It helps to map arrays accessed linearly in a specified loop to the core-local memory during the execution of the loop. Instead of mapping the whole array, the `stream` directive maps the shared array by windows. A window is a set of continuous array elements that are mapped to the core-local memory on the same core at the same time. In the first case, the loop consumes each window in sequence, as shown in Fig.6(a). For the second case, we extend the `omp parallel for` and `omp for` directives defined in the OpenMP specification to allow the `stream` directives as their suffix. OpenMDSP defines each window by array elements consumed in one chunk of the parallel loop. Correspondingly, OpenMDSP changes semantics of chunk size in OpenMP specification. The default chunk size is not a certain value but is determined at runtime as needed. OpenMDSP implementation should

always guarantee that the core-local memory is able to carry each window. If the chunk size is specified in the `schedule` clause but the corresponding window is too large to reside in the core-local memory, the chunk size specified by the user will be ignored. Fig.6(b) illustrates how the `stream` directive maps shared array to core-local memory window by window during the execution of a parallel loop.

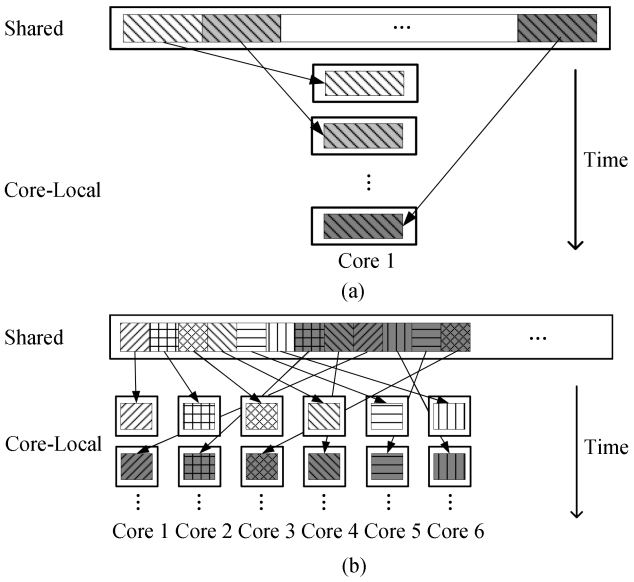


Fig.6. `stream` directive duplicates a large shared array window by window during the execution of a loop. (a) `stream` for a sequential loop. (b) `stream` for a parallel loop.

The syntax of the `stream` directive is similar to that of the `distribute` directive. Variables specified after `stream` should be arrays or pointers to array elements. These arrays must be accessed in a stream pattern during the following loop, and OpenMDSP will map the arrays to the core-local memory window by window. Programmers can use the `rate` clause to specify the number of elements accessed in a given iteration. The `copyin` and `copyout` clauses have the same semantics as those in the `distribute` directive, except that the copying-in or copying-out operations are performed before or after processing each window.

Fig.7 gives an example of the `stream` directive.

Although `distribute` and `stream` for parallel loops act in similar ways, their design goals differ. `stream` is useful for boosting the performance for arrays that are too big to distribute into the core-local memory. If an array is accessed in several parallel loops, `distribute` is a better choice.

Another advantage of the `stream` directive is the associated possibility for runtime to invoke the DMA controller for data transfer between the core-local memory and shared memory to hide latency, because most multi-core DSPs have DMA controllers, which copy data without any effort required of the DSP cores. For the `stream` directive, the OpenMDSP runtime should process each window by three steps, i.e., transferring the data to the core-local memory when `copyin` appears, processing the computation by the loop body, and copying the data out from the core-local memory when `copyout` appears. The OpenMDSP runtime can invoke the DMA controller for the first and last steps, i.e., when performing the first step in parallel with the computation of the previous window assigned to the current thread and when performing the last step in parallel with the computation of the next window.

```
 1 void complex_mul(int n, float* r, const float* a,
 2                  const float* b) {
 3   int i;
 4 #pragma omp parallel for
 5 #pragma domp stream(a, b) rate(2) copyin
 6 #pragma domp stream(r) rate(2) copyout
 7   for (i = 0; i < n; i++) {
 8     r[i*2] = a[i*2]*b[i*2] - a[i*2+1]*b[i*2+1];
 9     r[i*2+1] = a[i*2]*b[i*2+1] + a[i*2+1]*b[i*2];
10   }
11 }
```

Fig.7. Complex vector multiplication, given as an example of `stream` directive.

2.5 Discussion of the Design Decisions of the Extension

Most OpenMP directives are used for partitioning jobs into parallelizable portions and then synchronizing them. The feature used to control the storage and migration of data is simple, which can be summarized as data is either private to a thread or shared, and `firstprivate`, `lastprivate` and `copyprivate` can be used to migrate the whole portion of a shared variable from or to its private copy. This is sufficient for a general-purpose CPU, at least for a CPU with uniform memory access, because each bit in the memory, regardless of whether it is private or shared logically, is ultimately stored in the memory with the same latency to access, and the cache is managed by hardware with a cache-coherence protocol. However, for typical multi-core DSPs, the situation becomes significantly more difficult. Fast core-local memory must be treated as software-managed cache to achieve reasonable performance. The existing OpenMP directives are too simple to fulfill such requirements, so the extensions are necessary.

We extend OpenMP by introducing new directives with the `domp` prefix rather than by reusing several existing directives or using the `omp` prefix, because this approach results in good compatibility with standard

OpenMP compilers. OpenMDSP programs can be compiled by OpenMP compilers for general-purpose CPUs because `domp` pragmas are ignored. This design decision respects the same principle as OpenMP, which makes its programs compatible with C/C++ compilers without OpenMP support. Ultimately, it is not our goal to propose a revision for the OpenMP specification, because the extension is not suitable for general-purpose CPUs, which remains the target platform of most OpenMP programs. As a result, using a different directive prefix for extended directives is the most logical choice.

Another major decision exists concerning the choice of the OpenMP version. OpenMP 3.0 Specification was released in May 2008. The major new feature introduced in OpenMP 3.0 is the task construct. However, we chose to design OpenMDSP based on OpenMP 2.5 based on the following considerations.

1) Our underlying applications benefit little from task construct. The task construct benefits for parallelizable algorithms of irregular data structures. However, the data processed by our target application are either signal series or matrices, which are naturally organized by regular arrays.

2) The task construct suffers from more overhead on a multi-core DSP. Tasks created by task constructs can be scheduled to run on any idle thread. However, on a multi-core DSP, different threads are running on different cores. Because the stack resides in core-local memory, then to support task migration between threads, the runtime library must dump the task's stack to shared memory after creating or suspending a task, suffering high latency in the process by accessing the shared memory and making task no longer a light-weight construct.

3 Design and Implementation

3.1 Overview of FreeScale MSC8156

We have implemented OpenMDSP for FreeScale MSC8156, which is a typical high performance DSP with six cores. One chip of FreeScale MSC8156 consists of six DSP cores. The memory hierarchy consists of four levels:

• *L*1 *ICache and L*1 *DCache* are the first-level instruction cache and data cache and are private to each core. They are transparent to the software. Each is of 32 KB in size.

• *L*2 *Cache and M*2 *Memory* are the second level memory with a total size of 512 KB and are private to each core. This aspect of the system can be configured for division into the L2 cache and M2 memory. L2 is transparent to the software while M2 is addressable core-local memory.

• *M*3 *Memory* is on-chip shared memory with a size of 1 056 KB. The latency is larger than L1, L2 and M2.

• *Main Memory* is off-chip shared memory and is accessed by two DDR controllers; this access always occurs with the largest latency.

Programmers can decide whether the cache is enabled or disabled for specific linking sections. Moreover, like other typical DSPs, FreeScale MSC8156 does not provide hardware managed cache coherence among different cores.

FreeScale MSC8156 is equipped with a DMA controller with 16 high-speed bidirectional channels. It can transfer blocks of data to and from the M2 memory, the M3 memory, and the DDR controllers.

3.2 Overview of Compilation Process

Fig.8 shows the compilation process of an OpenMDSP application in OMDPFS. For an OpenMDSP source file named `a.c`, the driver of our OpenMDSP compiler first feeds the file into the C preprocessor (CPP). The output of CPP is the input of OMDPC, which is the source-to-source OpenMDSP compiler of OMDPFS. OMDPC transforms OpenMDSP directives into bare C code, and then the C code is fed into the Starcore C compiler (SCC) and the assembler (AS). SCC and AS are tools in FreeScale MSC8156 SDK. The output of AS is the relocatable object file `a.o`.

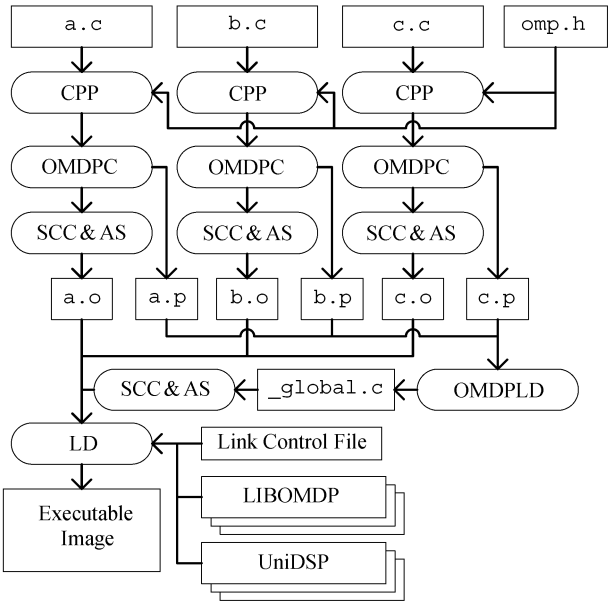


Fig.8. Compilation process of an OpenMDSP application in OMDPFS.

Additionally, OMDPC generates another file, `a.p`, which contains summary information of `a.c`. OMDPLD synthesizes these summary files and generates

`_global.c`, the global data file. We discuss the content of the global data file in Subsection 3.4.

The driver calls the linker to link all object files together with the LIBOMDP and UniDSP library, finally generating the executable image. LIBOMDP is the runtime library of OMDPFS, which is based on UniDSP. At runtime, each core loads the same executable image for execution, as a result of the SPMD nature of OpenMP.

UniDSP is a DSP operating system developed and internally used by Huawei Technologies Co. Ltd. with the goal to provide a uniform platform for high-level applications on different underlying hardwares. UniDSP takes over issues such as task management, memory management, synchronization, interrupt handling.

OMDPC and LIBOMDP are primary parts of OMDPFS; the header file `omp.h`, OMDPLD and the link control file also are provided by OMDPFS.

3.3 Overall Design

3.3.1 Implementing the Execution Model

In UniDSP, a task is the basic scheduling unit. A UniDSP task can only run on the core that creates it and never migrate to other cores. In OMDPFS, we map OpenMDSP threads to tasks on different cores.

For each core that loads the OpenMDSP image, we create one task to perform the job of one OpenMDSP thread. The entry of this task is a function in LIBOMDP. Because all cores execute the same image, LIBOMDP decides what to do based on the ID of the current core. The task running on the core with the least ID is treated as the master thread. In this task, the transformed entry function of the application is executed at startup. Tasks running on other cores pend on a semaphore until the master thread encounters a parallel region and post the semaphore to notify them.

3.3.2 Implementing the Memory Model

As stated before, OpenMDSP inherits the memory model of OpenMP, which is a relaxed-consistency, shared memory model. In general-purpose CPUs, because the cache coherence protocols guarantee sequential consistency for memory operations, which is a stricter design than relaxed-consistency, it is trivial to fulfill the OpenMP memory model.

FreeScale MSC8156 does not maintain cache coherence among the L1 and L2 cache of different cores. Instead, it provides instructions to invalidate the specified data in the cache. One intuitional method used to fulfill the relaxed-consistency model is the invalidation of data at each flush operation. However, the compiler cannot always analyze accurately what must be invalidated. Using a conservative analysis result usually causes the invalidation of all shared data, which is a time consuming operation.

Because of the considerations stated above, we disable the cache for shared sections to fulfill the memory model of OpenMP. Certainly, this action causes a decrease of the performance. Extensions of the distributed array and stream access can reduce the usage of shared memory, which makes the performance acceptable under most situations. In Section 4 we provide a measurement of the performance as affected by the cache.

3.4 Transformation Strategy

The OpenMP implementation method is well developed for the CPU[5]. Our transformation strategy inherits from the implementation on the CPU in many aspects, such as parallel region outlining, the transformation of work-sharing constructs, the synchronization constructs. In this subsection, we focus only on the differences and tricky points.

3.4.1 Transformation of Shared Data

As stated in Section 1, one critical problem is how shared variables with automatic storage duration are handled. In C language, by default, variables with automatic storage duration are placed on the stack, which resides in the core-local memory. To solve the problem, we create a shared stack on M3. OMDPFS places two kinds of variables with automatic storage duration on the shared stack, including:

• variables shared by at least one parallel region. OMDPFS puts these variables on a shared stack during its full life cycle. We call these variables the *permanent shared variables*.

• variables that are not shared by any parallel region, but are read or written by worker threads, for example, variables listed in `firstprivate` and `reduction` clauses. OMDPFS places these variables on the stack in the core-local memory initially, and copies its value to the shared stack before entering the parallel region (for `firstprivate` and `reduction`) or copies the value back to the private stack after exiting the parallel region (for `reduction`). We call these variables the *temporary shared variables*.

Fig.9 shows example code and illustrates the layout of private stacks and a shared stack. `a` is a private variable and only resides on private stacks. `b` appears in the `firstprivate` clause; as a result, it is a temporary shared variable and resides initially in the private stack of the master thread. Then OMDPFS copies its value to a temporary shared frame before the parallel region. Worker threads use its value to initialize its private ver-

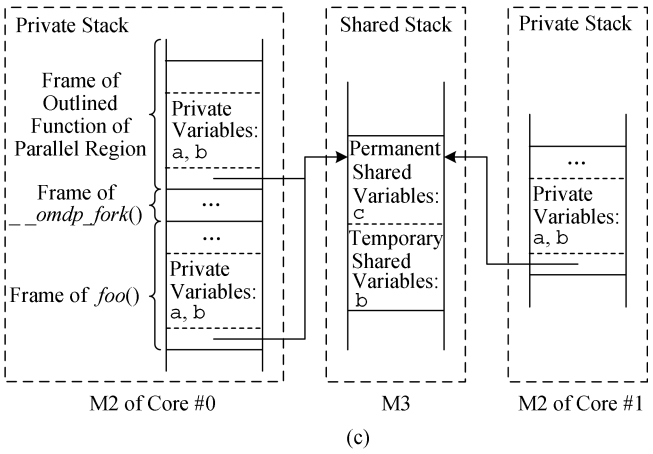
sion. `c` is a permanent shared variable that only resides on the shared stack. In the master thread, shared variables are accessed by the pointer to the current frame in the shared stack. The frame pointer of the shared stack is passed to worker threads during the execution of a parallel region, allowing the worker threads to access the variables shared by that pointer.

```
1 void foo() {
2   int a = 1, b = 2, c = 3;
3   ...
4 #pragma omp parallel private(a) firstprivate(b)
5   bar(&a, &b, &c);
6 }
```

(a)

```
1 typedef struct {
2   int b;
3   union {
4     struct {
5       int c;
6     } tmp1;
7   }
8 } __foo_shared_t;
9 void __foo_reg1(__foo_shared_t* __shared) {
10  int a;
11  int b = __shared->tmp1.b;
12  bar(&a, &b, &(__shared->c));
13 }
14 void foo() {
15  int a = 1, b = 2;
16  __foo_shared_t* __shared = (__foo_shared_t*)
17    __omdp_reg_share(sizeof(__foo_shared_t);
18  __shared->c = 3;
19  ...
20  {
21    __shared->tmp1.c = c;
22    __omdp_fork(&__foo_reg1, __shared);
23  }
24 }
```

(b)



(c)

Fig.9. Sample provided to show the transformation strategy of shared variables with automatic storage duration. (a) Source code. (b) Code after transformation. (c) Illustration of the shared data transformation.

3.4.2 *Transformation of* `distribute` *and* `stream` *Directives*

For each array or pointer specified in the `distribute` directive, a corresponding local pointer is created and initialized by an LIBOMDP function, `__omdp_distribute()`. All references of the original array or pointer inside the statement followed by the `distribute` directive are replaced by the local pointer.

`__omdp_distribute()` allocates the M2 memory for the local duplication of a given array. The value assigned to the local pointer is not the first address of the local duplication but the address that pretends to be the first element of the whole array in M2. As illustrated in Fig.10, `arr[]` is an array with 12 elements, and the main section for one thread of `arr[]` consists of `arr[6]` and `arr[7]`. The replacing pointer `local_arr` is assigned by the 6th element before the allocated main section, so elements in the main section can be accessed by its original index.

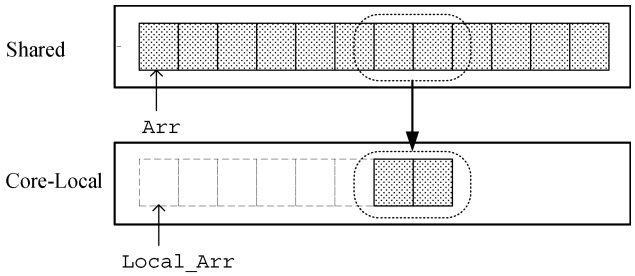


Fig.10. Illustration of the transformation for the distributed array.

The arrays and pointers specified in `stream` directives are handled in similar way. The runtime library determines the window size based on the available capacity of the M2 memory. Before executing each chunk of the parallel loop, the runtime library resets the local pointer to ensure references to elements based on subscripts within the current window falling into the right address in the M2 memory. The runtime library also copies elements in the current window from the shared memory to M2 before executing each chunk of the parallel loop, or the runtime library copies the elements back afterward, when either `copyin` or `copyout` clauses are present.

3.4.3 *Transformation of Data Placement Directives*

FreeScale MSC8156 SDK provides a "section" attribute, which is a kind of C extension, to map variables to the specified section. In OMDPFS, we predefine three sections in our link control file, which are placed in M2, M3 and DDR respectively. OMDPC deals with shared variables on M3 or DDR and thread-

private variables on M2 by inserting corresponding attributes. For the threadprivate variables on M3 or DDR, OMDPC replaces those variables with pointers on M2 which point to their original type. These pointers are initialized by the global data file.

3.4.4 Contents of Global Data File

The global data file contains the function `__omdp_global_initialize()`, which is called by LIBOMDP during initialization.

LIBOMDP provides a function for the allocation of memory on M3 and DDR for threadprivate variables and initializes these pointers. OMDPLD generates code in `__omdp_global_initializes()` to call this function to initialize these threadprivate variables.

OMDPC assigns one critical handle for each name of the critical region and generates code to call LIBOMDP functions for the `critical` directive with its critical handle. All critical handles are defined in the global data file, and `__omdp_global_initialize()` contains code to initialize these critical handles.

3.5 Runtime Library

Based on the transformation strategy discussed in Subsection 3.4, the implementation of LIBOMDP is an ordinary process. We write LIBOMDP in the C language with an extension provided by FreeScale MSC8156 SDK. LIBOMDP depends on UniDSP for low level operations.

LIBOMDP is responsible for managing the shared stack. The shared stack is implemented as a linked list of memory blocks. Memory blocks are allocated from the heap on M3. When pushing a new frame into the stack, LIBOMDP tries to allocate space in the last node. If the available space is insufficient, a new memory block is created and appended in the linked list. When a frame popping operation empties a block, the memory block is not freed immediately. LIBOMDP merges several empty blocks into a bigger block when any empty block is to be used again. After a few operations, the linked list tends to be stable and with little fragmentation, and pushing and popping operations become very efficient.

During a loop with the `stream` directive, LIBOMDP invokes the DMA controller to copy data between the core-local memory and shared memory. By default, LIBOMDP uses two different channels for each core to copy the data in and out respectively; as a result, up to 12 of the 16 DMA channels are reserved for LIBOMDP. LIBOMDP can also be configured to use only one channel for each thread or to disable the DMA for `stream` directives, in case these resources must be reserved for another purpose.

3.6 Source-to-Source Compiler

We adopt Cetus[6], a source-to-source compilation framework written in Java as our infrastructure. OMDPC is built based on the C parser and intermediate representation (IR) provided by Cetus. The C parser in Cetus is based on ANTLR[7], an $LL(k)$ parser generator. Additionally, Cetus provides an OpenMP parser that is written from scratch. The OpenMP parser uses a `String` to `String HashMap` to represent the clause names and values of OpenMP directives. However, it is difficult to analyze the semantics of a general expression represented as a `String`. For example, any of the `if` clauses in a `parallel` directive, the `schedule` clause in a `for` directive, and the `size`, `peek` and `bulk` clause in a `domp distribute` directive contain a general expression. A general solution requires ANTLR to parse the expressions inside the OpenMDSP clauses. Therefore, our compiler does not use the original OpenMP parser.

We provide a brief introduction for the construction of our source-to-source compiler.

3.6.1 IR of OpenMDSP Directives

We have extended the IR class hierarchy to introduce new kinds of IR node to represent OpenMDSP directives and clauses. We have created two abstract classes, `OmpDirective` and `OmpClause`, as the father classes of OpenMDSP directives and clauses. We have created one class for each kind of OpenMDSP directive by inheriting `OmpDirective`. For each group of OpenMP clauses with similar form, we have created one class for the group by inheriting `OmpClause`, for example, `private`, `firstprivate` and `lastprivate` clauses are represented by the class `OmpcVariableList`, and `ordered`, `nowait`, `copyin` and `copyout` are represented by the class `OmpcTag`. The expressions inside OpenMDSP clauses are represented by the subclasses of `Expression`, which is provided by Cetus to abstract any expression in plain C code. Representation by `Expression` makes the expressions inside OpenMDSP clauses easier to analyze.

3.6.2 Parser

We have extended the ANTLR grammar file of Cetus, added rules for OpenMDSP directives and created corresponding IR nodes.

3.6.3 Transformation Phases

The transformation of the OpenMDSP program consists of several phases.

• *OpenMDSP Normalization.* This phase transforms combined parallel work-sharing constructs into separate

parallel constructs and work-sharing constructs. For example, `parallel for` directives are transformed into a `parallel` directive enclosing a `for` directive. Additionally, this phase transforms the `sections` directives to `for` directives with a `switch` statement. This phase simplifies the jobs of the remaining phases because several kinds of OpenMDSP directives are eliminated.

• *Task Creation Function Replacement.* This phase replaces the call sites of the task creation function of UniDSP by `__omdp_task_create()`.

• *Parallel Region Outlining.* This phase outlines parallel regions to separate functions. Data sharing attributes are fulfilled by this phase as shown in Fig.9.

• *Threadprivate Transformation.* This phase transforms the variable declarations of threadprivate variables on M3 and DDR. Because the data type changes, this phase also traverses the whole IR to transform the reference points of these variables to pointer form.

• *Transformation of Other Directives.* This phase transforms all other directives. All directives remaining in this phase can be transformed in place. The `OmpDirective` class has an abstract method to obtain its IR in bare C form. This phase invokes this method for each OpenMDSP directive IR and then replaces it.

3.6.4 Framework for Data Type Analysis

The transformation of several directives requires the data types of some variables to be queried. Unfortunately, Cetus does not provide a unified representation of data type. The type of variable given can be only reasoned by analyzing the IR tree of the declaration, which is complex and messy. For example, it is subtle to reason the element type of a given array only by the IR of its declaration, because the element type may be a scalar type, array type, data pointer, function pointer or any combination of these types. What is worse, any data type may be defined by *typedef*, and querying a type may refer to several *typedef* declarations.

To solve such problem, we extend Cetus by introducing a unified representation of data type, as well as by constructing a conversion utility between data types and the IR of their declaration. The data type is represented as a tree in our framework, which is trivial for identifying the element type of an array, the target type of a pointer, each field type of a structure, etc. A type table is created for each source file.

4 Experimental Results

4.1 Benchmarks

Seven benchmarks, as shown in Table 2, are chosen to evaluate OMDPFS. Among the benchmarks, FFT, CE, MED and DQAM are kernels in the critical stages of the LTE base station uplink, as stated in Subsection 1.1. A normal LTE data size is chosen for the input and output datasets, both of which can be fitted into the M3 memory.

CVDP, MP and FIR are other kernels widely used in DSP. The input is designed to fit into the M3 memory. MPL and FIRL are similar to MP and FIR, respectively, but require memory capabilities beyond that of M3, thus DDR is used to hold most of the data.

All benchmarks are parallelized based on legacy sequential code. For each benchmark, no more than 10 OpenMDSP directives have been used. FIR and MP utilize the extended `distribute` and `for respect` directives. For benchmarks with large datasets, FIRL and MPL utilize the `stream` directive to improve the performance.

All benchmarks are compiled by OMDPFS and are optimized at the O2 level using SCC.

Table 2. List of Selected Benchmarks

ID	Application Name	Data Size	Lines of Code		
			SEQ	OMDP	MANU
FFT	Fast Fourier Transform	Signal length: 1 024	265	271	302
CE	Channel Estimation	Signal length: 1 024	113	119	147
MED	MIMO Equalizer & Decoder	Signal length: 1 024	147	157	190
DQAM	Demodulation of 64 Quadrature Amplitude Modulation	Signal length: 1 024	95	97	120
MP	Matrix Production	$64 \times 64 \times 64$	66	71	94
FIR	Finite Impulse Response Filter	Signal length: 2 048 response order: 256	71	76	101
MPL	Matrix Production (large dataset)	$512 \times 512 \times 512$	66	72	104
FIRL	Finite Impulse Response Filter (large dataset)	Signal length: 1 048 576 response order: 256	71	76	111
CVDP	Complex Vector Dot Production	Vector size: 4 096	68	72	98

Note: the number of source code lines is listed here. SEQ: sequential code, OMDP: OpenMDSP code, MANU: manually parallelized code.

4.2 Speedup

Fig.11 shows the speedup curves derived using 1 to 6 parallel threads, normalized with the run time of the serial versions with the cache enabled.

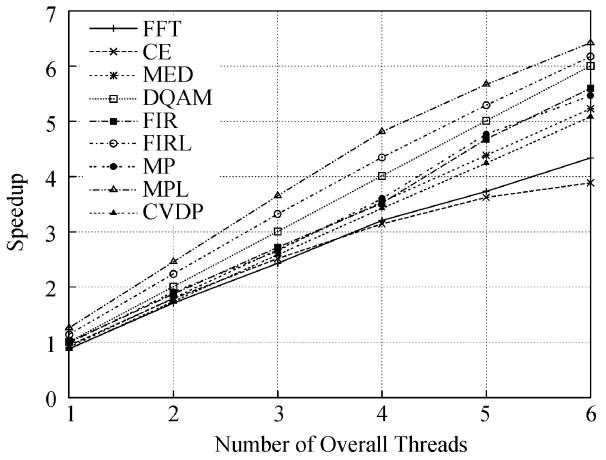


Fig.11. Speedup of each benchmark, normalized to the performance of a serial version with the cache enabled.

With one thread, the average speedup is 0.999, and the worse case speedup is 0.874. The two benchmarks that use the `stream` directive, identified as FIRL and MPL, get speedup more than one over the sequential version, because DMA hides the latency to access the data in DDR. In the sequential version, although the cache is enabled, it still suffers latency when loading data from the DDR to the cache. With 6 threads, 7 of 9 benchmarks have achieved a speedup of 5+. The other two, FFT and CE, have achieved speedup of 4.34 and 3.89, respectively.

The poor speedup of FFT is due to the irregular memory access pattern, which prevents the FFT algorithm from utilizing the distributed array. CE is constituted with several small loops, each of which consumes a small portion of the total run time. The overhead of management becomes dominant when more threads are used, leading to its relatively flat speedup curve.

Additionally, we manually parallelized all of these benchmarks and carefully tuned the performance to explore the potential of the parallelized code. The performance comparison between the two versions is shown in Fig.12. Using OMDPFS to compile, CE and CVDP suffer from the significant overhead incurred by the runtime library, which results in a 20% performance gap relative to the manual version. FFT also loses by 10% because it has a parallel inner loop, which produces noticeable loop management overhead. For some benchmarks (FIR and MP), OMDPFS wins the performance against its manual counterparts. This is caused by some subtle fluctuations that originate from the compiler and the underlying hardware. For example, OMDPFS outlines parallel regions, which can affect register allocation.

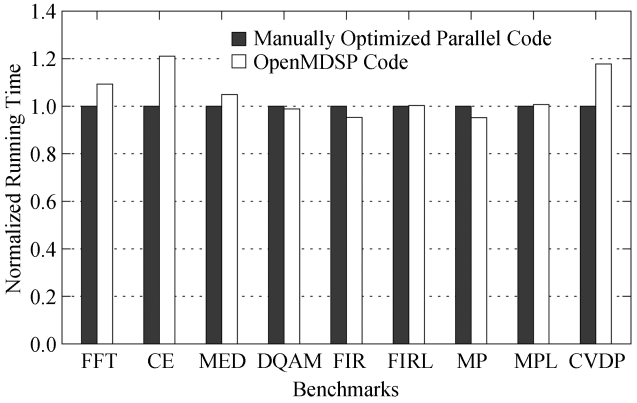


Fig.12. Comparison of OpenMDSP performance with a manually parallelized and optimized program running with 6 threads. The running time is normalized to the running time of manually parallelized programs of each benchmark.

We also have implemented the same benchmarks for a general-purpose CPU and measured their speedup for comparison. We tested their performance on a 4-way Quad-Core AMD Opteron^{TM} Processor 8347 server working at 1.9 GHz and built the benchmarks with an Intel C Compiler (ICC) version 11.1 with optimization level O2. As shown in Fig.13, the speedup of DQAM, FIRL, FIR, MPL and MP is not far behind the linear situations, but the speedup of the other four benchmarks does not perform well with respect to scalability. We have observed that all of these four benchmarks with poor performance have parallel constructs that are too fine grained. Their poor performance is due to the high cost in OpenMP runtime management related to the cost in real computation.

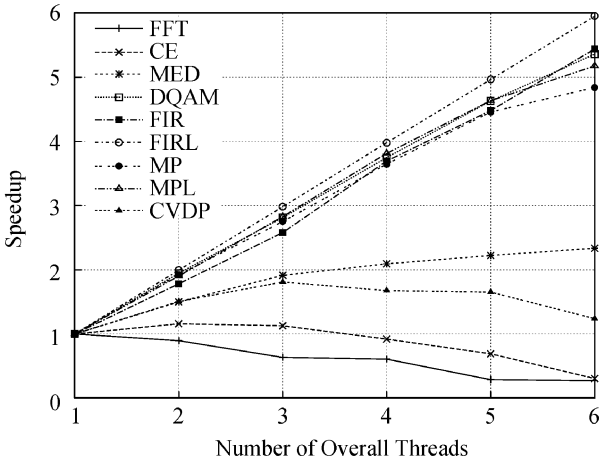


Fig.13. Speedup of each benchmark on a general-purpose CPU, normalized to the performance of a serial version with the cache enabled.

To study that quantitatively, we measured the running time of each application's serial version; additionally, we counted the number of OpenMP scheduling events, such as parallel regions, parallel loops and barriers. We obtained the frequency of OpenMP scheduling events by dividing the overall number of events by the running time. As shown in Table 3, the benchmarks with poor scalability actually have a significantly higher frequency of OpenMP scheduling events, which confirms the fact that a granularity of parallel constructs that is too fine does harm to the scalability on a general-purpose CPU. Although their OpenMDSP implementations also suffer from fine granularity, these benchmarks behave significantly better than the general-purpose CPU in terms of scalability. The result shows that our OpenMDSP implementation for the multi-core DSP has significantly less overhead than ICC's OpenMP implementation for the general-purpose CPU. In other words, it is more practical to exploit fine grain parallelism on the multi-core DSP using our OpenMDSP implementation.

Table 3. Occurrence of OpenMP Scheduling Events per Second

	`omp parallel`	`omp for`	Barrier
FFT	651 621.0	651 621.0	651 621.0
CE	774 398.0	2 323 195.0	774 398.0
MED	409 858.0	409 858.0	409 858.0
DQAM	20 357.0	20 357.0	20 357.0
FIR	9 279.0	9 279.0	9 279.0
FIRL	1.7	1.7	1.7
MP	37 447.0	37 447.0	37 447.0
MPL	0.5	0.5	0.5
CVDP	559 475.0	559 475.0	559 475.0

Note: `omp parallel for` construct is considered as an `omp for` construct nested in an `omp parallel` construct. All implicit barriers in OpenMP constructs are counted in the last column.

4.3 Effect of Extended Directives

In our benchmarks, several can use `distribute`, `for respect`, and `stream` directives when writing parallel code. Fig.14 compares the performance with and without these directives. Among all of the benchmarks, FIR and MP benefit the most from the distributed array because they can achieve excessive reuses of the array elements. In MED, each element of the distributed array is only used twice, thus MED benefits less from the `distribute` and `for respect` directives. The distributed array harms the performance of CVDP because there is no reuse for each element. The `stream` directive brings significant performance improvements for FIRL and MPL; this improvement also can be credited to the array element reuses.

For FIRL and MPL, we also compared the performance between enabling and disabling the DMA for the `stream` directive, and the results indicate that the use of DMA to hide the latency of data transfer is a signifi-

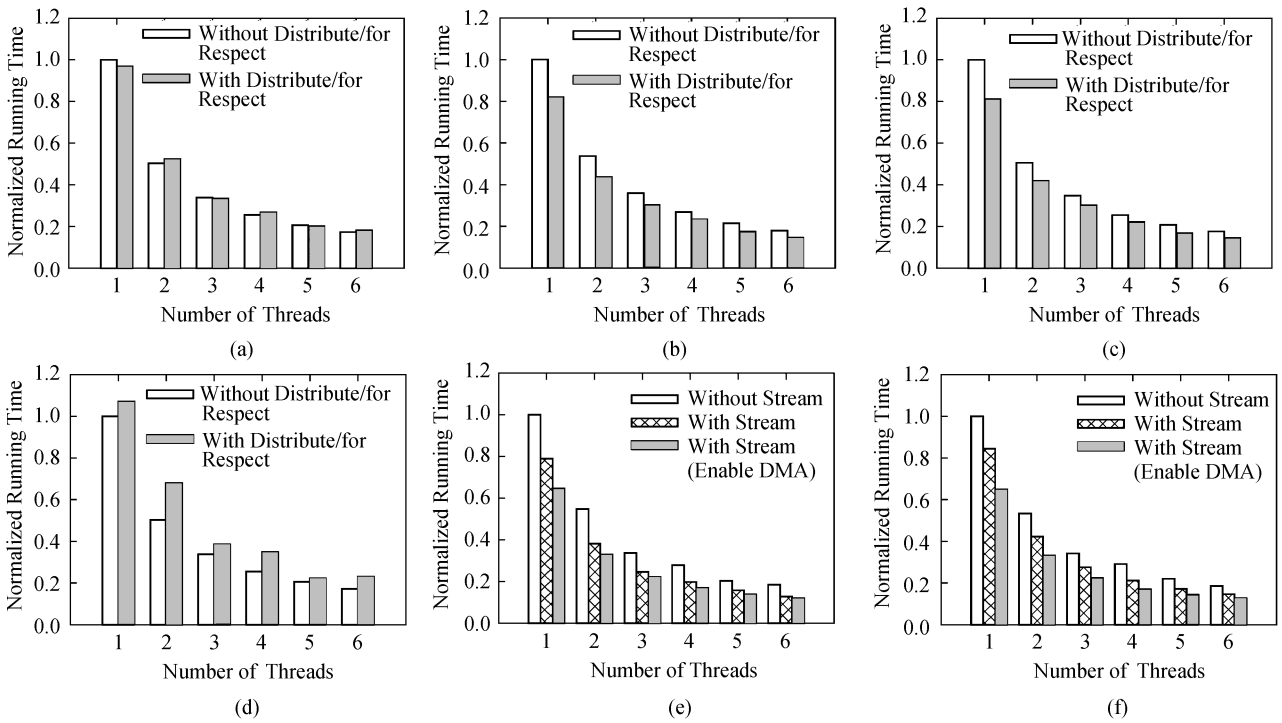


Fig.14. Comparison of the running time based on varying the usage of the distributed array or stream. The running time is normalized to one thread without the use of any extension. (a) MED. (b) FIR. (c) MP. (d) CVDP. (e) FIRL. (f) MPL.

cant benefit. When the number of threads is low, the performance gained by enabling DMA is more pronounced. This can be explained by the fact that when more threads are running, the memory access by the additional DMA channels approaches the bandwidth of the DDR; as a result, some channels are more frequently obligated to wait for others before accessing the data in the DDR, and sometimes the latency of data transfer cannot be hidden thoroughly.

Overall, the extended `distribute`, `for respect`, and `stream` directives are beneficial for applications with reusable arrays, and `stream` works better by enabling DMA for the transfer of data between the core-local memory and DDR. However, with little or no data reuse, the opposite effects also can occur.

5 Limitation and Discussion

In this paper, we use OpenMDSP to express data parallelism, which is both intra-task and low-level. However, OpenMDSP is not applicable for inter-task high-level parallelism, such as task parallelism and pipeline parallelism. Currently, we are still using vendor SDK to support task or pipeline parallelism, which is tedious, error-prone, and unportable.

Because intra-task parallelism and inter-task parallelism exist at different levels in the system, the problem of intra-task and inter-task parallelism are orthometric in some sense. We believe that it is a poor choice to use OpenMP for inter-task parallelism, because OpenMP does not provide any support for pipeline parallelism; moreover, compared with inter-task algorithms, it is significantly less possible for inter-task workflows to reuse the OpenMP code written for a general-purpose CPU. In our future work, it is our goal to support inter-task parallelism in an easy and portable way, using OpenMDSP procedures as basic building blocks.

6 Related Work

OpenMP is an industry standard parallel programming model. During the last decade, it has been implemented in most mainstream compilers for general-purpose CPUs such as GCC②, Intel C Compiler[8], Open64[5], and Microsoft Visual C++. Researchers also investigated techniques for the optimization of OpenMP implementation on general-purpose CPUs[8-11]. OpenMP implementation technology is well developed for general-purpose CPUs.

OpenMP also has been implemented on some architectures other than general-purpose CPUs. There are several OpenMP implementations for Cell[12-13], GPGPUs[14-15] and Software Scalable System on Chip(3SoC)[16-17] architectures. OpenMP is extended in these researches to support both complex memory hierarchies and system heterogeneity. With respect to the memory hierarchy directives, the *data placement* directive in OpenMDSP shares some common ideas with the data mapping clause in [15], but the *distributed array* and *stream access* directives are unique in comparison with these researches.

OpenMP 4.0 specification, which has just been released, introduces a new class of constructs, i.e., device constructs. With device constructs, OpenMP programs running on CPU can offload code and data to other computing devices such as GPGPU and Xeon Phis. It gives a standard way to coordinate different kinds of processors and corresponding memory, but does not address the problem of utilizing complicated memory hierarchies for the same processor.

Many other programming models also have been investigated for DSP and other Multi-Processor Systems-On-Chip (MPSoC) architectures, such as StreamIt[4], SoC-C[3], OSCAR[18-19] and the MPSoC Application Programming Studio (MAPS)[20]. We discussed the strengths and weaknesses of StreamIt in Subsection 1.2 so we skip additional discussion of StreamIt here. SoC-C[3], OSCAR and MAPS[20] are all good at task/pipeline parallelism support but lack high-level abstractions for data parallelism. We think OpenMDSP is complementary to these researches with respect to the expression of data parallelism inside a task.

Regarding the portability of a programming model, researchers also have proposed a retargetable parallel programming framework for MPSoC[21]. Researchers have designed a common intermediate code (CIC) and developed a framework to map task codes to CIC. Investigators have used XML file to describe the relations between tasks, and both Message Passing Interface (MPI) and OpenMP can be used in a task. Currently, the framework translates OpenMP to MPI code.

The effort made by OpenMDSP to distribute data into the core-local memory looks similar to High Performance Fortran (HPF)[22] and Partitioned Global Address Space (PGAS) programming models such as UPC (Unified Parallel C)[23] and Co-Array Fortran[24], which also deal with the problem of distributed memory, although these approaches possess some fundamental differences. The memory of target platforms of HPF and PGAS is composed of chunks of local memory. In other words, the sum of all local memories is the whole memory. As a result, any data ultimately resides on local memory, and HPF and PGAS provide some way to express how the data are permanently distributed among

②GOMP: An OpenMP implementation for GCC. http://gcc.gnu.org/projects/gomp, Nov. 2013.

the different chunks of local memory. However, the memory of a multi-core DSP is composed of several levels, and the sum of the core-local memory is only a small portion of the whole memory. Most data must ultimately reside on the large shared memory. Local memory can only cache these data temporarily for better performance, which is the purpose of the distributed array and stream access extensions of OpenMDSP.

7 Conclusions

Programming multi-core DSP systems is important yet challenging. The key problem we addressed in this paper is dealing with core-local memory and non-cache coherent shared memory with high-level directives. Our design and implementation show that these memory hierarchies can be managed effectively with a few extensions to the OpenMP 2.5 standard. OpenMDSP performs well at parallel programming inside tasks, forming a solid foundation for our future work on inter-task parallel programming. We expect that this work will motivate more investigations on programming these memory hierarchies which is critical to the success of future multi/many-core systems.

Acknowledgments We thank De-Hao Chen, Ji-Dong Zhai, and Tian-Wei Sheng for their insightful comments. Finally, we would thank Zi-Ang Hu, Qian Tan, and Li-Bin Sun for their help on experiments. We also thank the anonymous reviewers for their constructive suggestions.

References

[1] Karam L, AlKamal I, Gatherer A, Frantz G, Anderson D, Evans B. Trends in multicore DSP platforms. *Signal Processing Magazine, IEEE*, 2009, 26(6): 38-49.

[2] Zyren J. Overview of the 3GPP long term evolution physical layer, 2007. http://www.freescale.com/files/wireless_comm/doc/white_paper/3GPPEVOLUTIONWP.pdf, Nov. 2013.

[3] Reid A D, Flautner K, Grimley-Evans E, Lin Y. SoC-C: Efficient programming abstractions for heterogeneous multicore systems on chip. In *Proc. the 2008 CASES*, October 2008, pp.95-104.

[4] Thies W, Karczmarek M, Amarasinghe S. StreamIt: A language for streaming applications. In *Proc. Int. Conf. Compiler Construction*, April 2002, pp.179-196.

[5] Liao C, Hernandez O, Chapman B, Chen W, Zheng W. OpenUH: An optimizing, portable OpenMP compiler: Research Articles. *Concurrency and Computation: Practice & Experience*, 2007, 19(18): 2317-2332.

[6] Dave C, Bae H, Min S, Lee S, Eigenmann R, Midkiff S. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 2009, 42(11): 36-42.

[7] Parr T, Quong R. ANTLR: A predicated-$LL(k)$ parser generator. *Software – Practice & Experience*, 1995, 25(7): 789-810.

[8] Tian X, Girkar M, Shah S *et al.* Compiler and runtime support for running OpenMP programs on Pentium- and Itanium-architectures. In *Proc. the 17th Parallel and Distributed Processing Symposium*, April 2003, pp.9-18.

[9] Müller M S. Some simple OpenMP optimization techniques. In *Lecture Notes in Computer Science 2104*, Eigenmann R, Voss M, (eds.), Springer, 2001, pp.31-39.

[10] Tian X, Girkar M, Bik A, Saito H. Practical compiler techniques on efficient multithreaded code generation for OpenMP programs. *Computer Journal*, 2005, 48(5): 588-601.

[11] Chapman B M, Huang L. Enhancing OpenMP and its implementation for programming multicore systems. In *Proc. Parallel Computing: Architectures, Algorithms and Applications*, September 2007, pp.3-18.

[12] O'Brien K, O'Brien K M, Sura Z *et al.* Supporting OpenMP on cell. *Int. J. Parallel Programming*, 2008, 36(3): 289-311.

[13] Wei H, Yu J. Loading OpenMP to Cell: An effective compiler framework for heterogeneous multi-core chip. In *Proc. the 3rd International Workshop on OpenMP*, June 2007, pp.129-133.

[14] Lee S, Min S, Eigenmann R. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In Proc. *the 14th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, Feb. 2009, pp.101-110.

[15] Lee S, Eigenmann R. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proc. the 2010 Conf. High Performance Computing Networking, Storage and Analysis*, Nov. 2010.

[16] Liu F, Chaudhary V. Extending OpenMP for heterogeneous chip multiprocessors. In *Proc. the 32nd International Conference on Parallel Processing*, October 2003, pp.161-168.

[17] Liu F, V. Chaudhary. A practical OpenMP compiler for system on chips. In *Lecture Notes in Computer Science 2716*, Voss M (ed.), Springer, 2003, pp.54-68.

[18] Kimura K, Mase M, Mikami H *et al.* OSCAR API for real-time low-power multicores and its performance on multicores and SMP servers. In *Lecture Notes in Computer Science 5898*, Gao G, Pollock L, Cavazos J, Li X (eds.), Springer, 2009, pp.188-202.

[19] Hayashi A, Wada Y, Watanabe T *et al.* Parallelizing compiler framework and API for power reduction and software productivity of real-time heterogeneous multicores. In *Lecture Notes in Computer Science 6548*, Cooper K, Mellor-Crummey J, Sarkar V (eds.), Springer, 2010, pp.184-198.

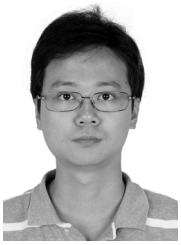
[20] Leupers R, Castrillón J. MPSoC programming using the MAPS compiler. In *Proc. the 15th Asia and South Pacific Design Automation Conference*, January 2010, pp.897-902.

[21] Kwon S, Kim Y, Jeun W, Ha S, Paek Y. A retargetable parallel-programming framework for MPSoC. *ACM Trans. Design Autom. Electr. Syst.*, 2008, 13(3): Article No.39.

[22] Kennedy K, Koelbel C, Zima H P. The rise and fall of High Performance Fortran: An historical object lesson. In *Proc. the 3rd ACM SIGPLAN Conf. History of Programming Languages*, June 2007, Article No. 7.

[23] El-Ghazawi T, Carlson W, Sterling T *et al.* UPC: Distributed Shared Memory Programming. Wiley-Interscience, 2003.

[24] Numrich R W, Reid J. Co-array Fortran for parallel programming. *ACM Fortran Forum*, 1998, 17(2): 1-31.



Jiang-Zhou He received the B.S. degree in computer science from Tsinghua University in 2007. Now he is a Ph.D. candidate in computer science and technology in Tsinghua University. His research interests include parallel and distributed computing, compiler technology, and programming models. He is a student member of CCF.

Wen-Guang Chen received the B.S. and Ph.D. degrees in computer science from Tsinghua University in 1995 and 2000 respectively. He was the CTO of Opportunity International Inc. from 2000 to 2002. Since January 2003, he joined Tsinghua University. He is now a professor and vice dean of the Department of Computer Science and Technology, Tsinghua University. His research interests include parallel and distributed computing, programming model, and mobile cloud computing. He is a member of CCF, ACM and IEEE.

Guang-Ri Chen received the B.S. and M.S. degrees in computer science from Xidian University in 1998 and 2003 respectively. He worked in Potevio Institute of Technology Co. Ltd. from 2003 to 2008. Since 2008 he works at Huawei Technologies Co. Ltd. His work focuses on LTE base station technologies.

Wei-Min Zheng received the master's degree from Tsinghua University in 1982. He is now a professor in the Department of Computer Science and Technology at Tsinghua University. His research interests include parallel and distributed computing, compiler technique, grid computing, and network storage. He is a fellow of CCF and a member of ACM and IEEE.

Zhi-Zhong Tang received the B.S. degree from Tsinghua University in 1970. He is now a professor in the Department of Computer Science and Technology at Tsinghua University. His research interests include compiler technique and CMP cache optimization. He is a member of CCF.

Han-Dong Ye is a research engineer of compiler and SDK tools in FutureWei, US R&D Center of Huawei. He has more than 10 years experience in compiler industry, and his research interests include programming language, compiler optimization, parallelization and runtime system.