A Practical OpenMP Compiler for System on Chips

Feng Liu[1] and Vipin Chaudhary[2]

[1] Department of Electrical & Computer Engineering, WSU, USA
fliu@ece.eng.wayne.edu
[2] Institute for Scientific Computing, WSU and Cradle Technologies, Inc.
vipin@wayne.edu

Abstract. With the advent of modern System-on-Chip (SOC) design, the integration of multiple-processors into one die has become the trend. By far there are no standard programming paradigms for SOCs or heterogeneous chip multiprocessors. Users are required to write complex assembly language and/or C programs for SOCs. Developing a standard programming model for this new parallel architecture is necessary. In this paper, we propose a practical OpenMP compiler for SOCs, especially targeting *3SoC*. We also present our solutions to extend OpenMP directives to incorporate advanced architectural features of SOCs. Preliminary performance evaluation shows scalable speedup using different types of processors and effectiveness of performance improvement through optimization.

1. Introduction

OpenMP is an industrial standard [1, 2] for shared memory parallel programming with growing popularity. The standard API consists of a set of compiler directives to express parallelism, work sharing, and synchronization. With the advent of modern System-on-Chip (SOC) design, chip multiprocessors (CMP) have become a new shared memory parallel architecture. Two major shared memory models exist today: Symmetric Multiprocessor machines (SMP) and distributed memory machines or clusters. Unlike these two models, SOCs incorporate multiple distinct processors into one chip. Accordingly, it has a lot of new features which normal OpenMP standard could not handle, or at least could not take advantage of.

By far there are no standard programming paradigms for SOCs or heterogeneous chip multiprocessors. Users are required to write complex assembly language and/or C programs for SOCs. It's beneficial to incorporate high-level standardization like OpenMP to improve program effectiveness, and reduce the burden for programmers as well. For parallel chips like *Software Scalable System on Chip (3SoC)* from Cradle, parallelism is achieved among different types of processors; each processor may have different instruction sets or programming methods. Thus, developing a standard parallel programming methodology is necessary and challenging for this new architecture.

Cradle's *3SoC* is a shared-address space multi-processor SOC with programmable I/O for interfacing to external devices. It consists of *multiple processors* hierarchically connected by two levels of buses. A cluster of processors called a Quad is connected

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 54–68, 2003.
© Springer–Verlag Berlin Heidelberg 2003

by a local bus and shares local memory. Each Quad consists of four RISC-like processors called Processor Elements (PEs), eight DSP-like processors called Digital Signal Engines (DSEs), and one memory Transfer Engine (MTE) with four Memory Transfer Controllers (MTCs). The MTCs are essentially DMA engines for background data movement. A high-speed global bus connects several Quads. The most important feature for parallel programming is that *3SoC* provides 32 semaphore hardware registers to do synchronization between different processors (PEs or DSEs) within each Quad and additional 64 global semaphores [3].

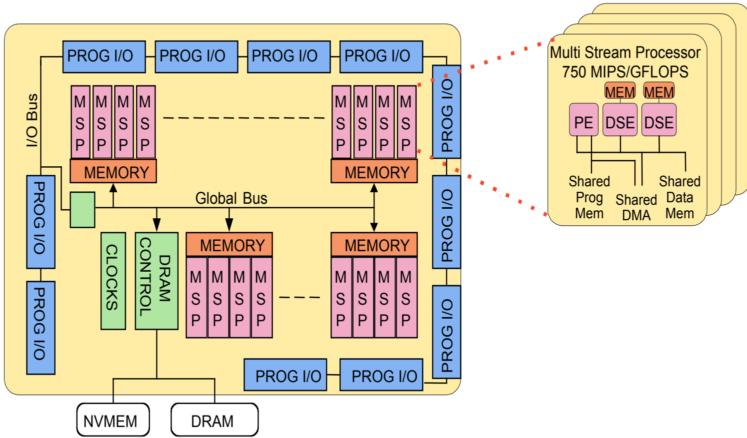

Fig. 1. *3SOC* Block Diagram

In this paper, we describe the design and implementation of our preliminary OpenMP compiler/translator for *3SoC*, with detailed focus on special extensions to OpenMP to take advantage of new features of this parallel chip architecture. These features are commonplace among SOCs; techniques used here may be targeted for other SOCs as well. Giants like Intel, Sun and IBM have addressed their own plans for SOCs and it's believed that CMP will dominate the market in the next few years. We give detailed explanation of each extension and how it should be designed in OpenMP compiler.

In the next section we briefly introduce the parallel programs on *3SoC,* targeting different processors: PEs or DSEs, respectively. In section 3 we present the design of our OpenMP compiler/translator with special focus on synchronization, scheduling, data attributes, and memory allocation. This is followed in section 4 with a discussion on our extensions to OpenMP. Section 5 outlines the implementation aspects of a practical OpenMP compiler for SOCs followed by a performance evaluation in section 6. Conclusion is given in section 7.

2. Parallel Programs on *3SoC*

In this section, we briefly describe the approach to program *3SoC*. Our OpenMP translator will attempt to create such parallel programs.

2.1 Programming Different Parallel Processors

A *3SOC* chip has one or more Quads, with each Quad consisting of different parallel processors: four PEs, eight DSEs, and one Memory Transfer Engine (MTE). In addition, PEs share 32KB of instruction cache and Quads share 64KB of data memory, 32KB of which can be optionally configured as cache. Thirty-two semaphore registers within each quad provide the synchronization mechanism between processors. Figure 2 shows a Quad block diagram. Note that the Media Stream Processor (MSP) is a logical unit consisting of one PE and two DSEs. We will now have a look at how to program parallel processors using PEs or DSEs.

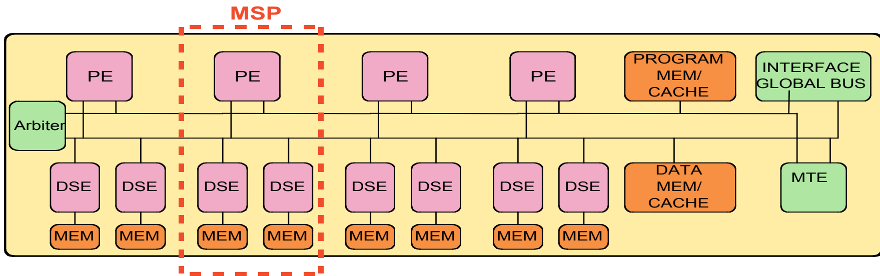

Fig. 2. Quad block diagram

2.2 Programming Using PEs

PE has a RISC-like instruction set consisting of both integer and IEEE floating point instructions. In *3SoC* architecture, there are a number of PEs which can be viewed as several "threads" compared with normal shared memory architecture. Each processor has its own private memory stack, similar to "thread" context. At the same time, each processor is accessing the same blocks of shared local memory inside Quad, or SDRAM outside Quad. These memories are all shared. In a typical *3SoC* program, PE0 is responsible for initializing other processors like PEs or DSEs, so that PE0 acts as the "master" thread while other processors act as "child" threads. Then PE0 will transfer parameters and allocate data movement among different processors. It will load MTE firmware and enable all MTCs. Through data allocation PE0 tells each processor to execute its own portion of tasks in parallel. PE0 will also execute the region itself as the master thread of the team. Synchronization and scheduling must be inserted properly. At the end of each parallel region, PE0 will wait for other processors to finish and collect required data from individual processor.

PEs are programmed using standard ANSI C. The *3SoC* chip is supplied with GNU-based optimizing C-compilers, assemblers, linkers, debuggers, and performance accurate simulators (refer to *3SoC* programmer's guide [4]). To incorporate several PEs to work in a parallel program, the approach is similar to conventional parallel programming [6].

We present the concept of translating an OpenMP program to a *3SoC* parallel program using several PEs. The *#pragma omp parallel* defines a parallel region whereas the directive *#pragma omp single* specifies that only one thread executes this scope. Correspondently, in the *3SoC* main program, each PE is associated with its ID, *my_peid*. The parallelism is started by PE0 *(my_peid==0)*, and only PE0 allocates and initializes all other processors. Once started, all PEs will execute *function()* in parallel. Within its own function context, each PE will execute its portion of tasks by associated processor ID. At the end of each parallel region, all PEs reach *an implicit barrier* where PE0 waits and collects data from other PEs. Finally, PE0 terminates other PEs and releases resources to the system.

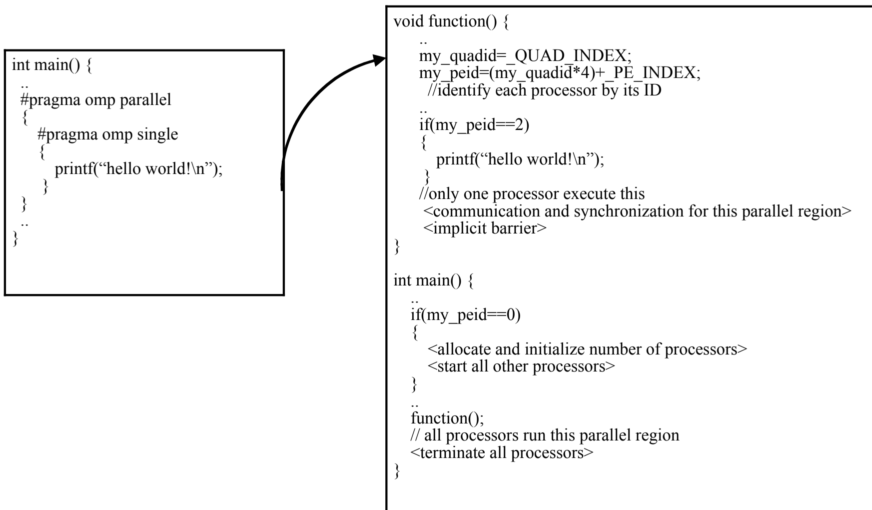

Fig. 3. Translation of a simple OpenMP program (left) to 3SoC parallel region (right)

2.3 Programming Using DSEs

DSE is a DSP-like processor which uses a different programming methodology. DSEs are programmed using C-like assembly language ("CLASM") combined with standard ANSI C. DSEs are the primary processing units within *3SoC* chip. Compilers, assemblers, and tools are supplied for DSE.

Writing OpenMP program for DSE requires a completely different approach. The controlling PE for a given DSE has to load the DSE code into the DSE instruction memory. Then PE initializes the DSE DPDMs (Dual Ports Data Memory) with desired variables and starts the DSE. The PE either waits for the DSE to finish by poll-

ing, or can continue its work and get interrupted when the DSE has finished its task. A number of DSE library calls are invoked. See Fig. 4.

First, the PE initializes the DSE library call via *dse_lib_init(&LocalState)*. Then the PE does some Quad I/O check and data allocation such as assigning initial values for the matrix multiplication. In the next for-loop, the PE allocates a number of DSEs and loads the DSE code into the DSE instruction memory by *dse_instruction_load()*. This is done by allocating within one Quad first, *dse_id[I]=dse_alloc(0),* if failed, it will load from other Quads. Afterwards, the PE loads the DPDM's onto the allocated DSEs, dse*_loadregisters(dse_id)*. Each DSE starts to execute the parallel region from the 0th instruction, via *dse_start(dse_id[i],0)*. PE is responsible for waiting for DSEs to complete computation and terminating all resources.

```
void main() {
        ..
        int dse_id[NUM_DSE];
        dse_lib_init(&LocalState);
        pe_in_io_quad_check();
        ..
        <Data allocation>
        ..
        // load the MTE firmware and enable all MTCs
        _MTE_load_default_mte_code(0x3E);

        for(i = 0; i < NUM_DSE; i++) {
                // allocate a DSE in this Quad
                dse_id[i] = dse_alloc(0);

                if(dse_id[i] < 0) {
                        // allocate DSE from any Quad
                         dse_id[i] = dse_alloc_any_quad(0);
                         if(dse_id[i] < 0) {
                           <error condition>
                         }
                }
                // load the instructions on the allocated DSEs
                dse_instruction_load(dse_id[i], (char *)&dse_function, (char
*)&dse_function_complete, 0);
        }
        // Load the Dpdm's on the allocated DSEs
        DSE_loadregisters(dse_id);

        for(i = 0; i < NUM_DSE; i++) {
                // Start all DSEs from the 0th instruction
                dse_start(dse_id[i], 0);
        }

        for(i = 0; i < NUM_DSE; i++) {
                // Wait for the DSE's to complete and free DSEs
                dse_wait(dse_id[i]);
        }
        <other functions>
        ..
        dse_lib_terminate();
        ..
}
```

Fig. 4. Sample 3SoC parallel program using multiple DSEs

3. Design of OpenMP Compiler/Translator

To target the OpenMP compiler for *3SoC*, we have to cope with the conceptual differences from standard OpenMP programs. OpenMP treats the parallelism at the granularity of parallel regions, and each function may have one or more independent parallel regions; while *3SoC* treats the parallelism at the level of function, where each

private data structure is defined within one function. We also have to treat the data scope attributes between processors like *"Firstprivate", "Reduction"* along with appropriate synchronization and scheduling.

3.1 Synchronization

Hardware semaphores are the most important features that distinguish normal chip multiprocessors from "parallel" chips. Developers can use hardware semaphores to guard critical sections or to synchronize multiple processors. On *3SoC* platform, each Quad has 32 local and 64 global semaphore registers that are allocated either statically or dynamically. For parallel applications on *3SoC*, the semaphore library *(Semlib)* procedures are invoked for allocating global semaphore or for locking and unlocking.

Equipped with this feature, users can implement two major synchronization patterns.

1. By associating one hardware semaphore for locking and unlocking, user can define a *critical* construct which is mutually exclusive for all processors.
2. By combining one semaphore along with global shared variables, OpenMP *barrier* construct can be achieved across all processors. Sample *barrier* implementation is as follows:

```
semaphore_lock(Sem1.p);
done_pe++;       //shared variable
semaphore_unlock(Sem1.p);
while(done_pe<(NOS)); //total # of parallel processors
_pe_delay(1);
```

For OpenMP compiler, *barrier* implementation is important. There are a number of *implicit* barriers existing at the end of each OpenMP parallel region or work-sharing construct. Therefore, hardware semaphores allocation and de-allocation becomes a crucial factor. We take two steps to implement this. First step is to allocate all semaphores into local shared memory as "_SL" to improve data locality, i.e. *"static semaphore_info_t SemA _SL"*. (Type of variables like "_SL" is discussed in Section 3.3). Secondly we allocate semaphores dynamically at run-time. This provides more flexibility than static approach with respect to the limited number of semaphores in each chip. Each semaphore is initialized and allocated at the beginning of the parallel region; while after use, it's de-allocated and returned to the system for next available assignment. This is done by system calls at run-time [4].

3.2 Scheduling and Computation Division

In OpenMP programs, the computation needs to be divided among a team of threads. The most important OpenMP construct is the work-sharing construct, i.e., *for*-construct. The parallelism is started by the OpenMP directive *#pragma omp parallel*. Any statement defined within this *parallel* directive will be executed by each thread of the team. For the work sharing constructs, all statements defined within will be di-

vided among threads. Each thread will execute its own share, like *for*-construct, which is a small portion of loop iterations.

A common translation method for parallel regions employs a *micro-tasking* scheme, like OdinMP [5]. Execution of the program starts with a *master thread*, which during initialization creates a number of *spinner threads* that *sleep* until they are needed. The actual task is defined in *other threads* that are waiting to be called by the spinner. When a parallel construct is encountered, the master thread wakes up the spinner and informs it of the actual task thread to be executed. The spinner then calls the task thread to switch to specific code section and execute. We follow this scheme but take different approach. For CMP, each *"thread"* is a processor, which has its own processing power and doesn't wait for resources from other processors. The number of "threads" is the actual number of processors instead of a team of *virtual threads*. It's not practical to create two "threads", one for spinning, and another for actual execution. In our implementation, we assign each parallel region in the program with a unique identifying function, like *func_1()*. The code inside the parallel region is moved from its original place and replaced by a function statement, where its associated region calls this function and processors with correct IDs execute selected statements in parallel.

Scheduling is an important aspect of the compiler. Conceptually there are three approaches to distribute workload, known as: *cyclic, block,* and *master-slave* methods. Cyclic and block methods use static approach while master-slave method distributes dynamically. To implement common OpenMP work-sharing constructs like *for-loop*, we apply different methods.

First, we try to use the static approach. The OpenMP compiler will analyze and extract the for loop header to find the index, loop initial, loop boundary and the increment, then distributes the workload into small slices. Each processor will be assigned a small portion of loop iterations based on algorithms like cyclic or block allocation. It is the responsibility of the programmer to ensure that the loop iterations are independent. Secondly, we deploy dynamic scheduling. For Master-slave method, it needs one or two master threads which are responsible for distributing workload and several slave threads which fetch their own portions from the master thread dynamically. The master thread will be created at the beginning of parallel region and assign workload to slave threads in *request-grant* manner. In our implementation for CMP, we didn't create a master thread due to limited number of processors and potential waste of resources. Instead, we use a global shared variable which is accessible to all processors. The total amount of work is evenly divided into small slices; each time a processor obtains one slice by accessing and modifying the shared variable. With the growing value of this shared variable, workload is distributed dynamically at runtime. A semaphore, which guarantees that only one processor can modify a variable at any given time, protects this shared variable.

We implement both *dynamic* and *static* scheduling on *3SoC*. User can choose one of the two scheduling methods through OpenMP compiler parameter during compilation. This provides more flexibility to do performance evaluation for different applications. Among the above two approaches, static allocation shows better performance on average. The reason is that *3SOC* is a CMP environment where each "thread" is a processing unit. By contacting the master thread or accessing the shared variable a

dynamic approach involves more synchronizations that interrupt the processors more frequently. Sample OpenMP code using static scheduling is shown in Fig. 5.

```
#pragma omp parallel shared(matA) private(k,j)
{
 for(k=0;k<SIZE;k++)
 {
  …
  #pragma omp for private(i)
     for(i=k+1;i<SIZE;i++)
         for(j=k+1;j<SIZE;j++)
                matA[i][j]=matA[i][j]-matA[i][k]*matA[k][j];
  }
} /* end of parallel construct */
```

```
{
    for(k=0;k<SIZE;k++)  {
        …
        {
         int count_ums=0,mymin,mymax;
         count_ums=((SIZE) - (k+1))/PES;

         mymin=my_peid*count_ums + (k+1);
         if(my_peid==(PES)-1)
            mymax=SIZE;
         else
            mymax=mymin+count_ums-1;

        for(i=k+1;i<SIZE;i++)
         if(i>=mymin && i<=mymax)
        {
              for(j=k+1;j<SIZE;j++)
              matA[i][j]=matA[i][j]-matA[i][k]*matA[k][j];
        }

        //barrier using hardware semaphore
         semaphore_lock(Sem2.p);
         done_pe2++;
         semaphore_unlock(Sem2.p);
         while(done_pe2<(PES));
            _pe_delay(1);
          if(my_peid==0)
            done_pe2=0;
        }
    }
    …
} /* end of parallel construct */
```

Fig. 5. Sample OpenMP code (LU decomposition) using static allocation

3.3 Data Attributes and Memory Allocation

In OpenMP, there are a number of clauses to define data attributes. Two major groups of variables exist: shared or private data. By default, all variables visible in a parallel region are shared among the threads. OpenMP provides several directives to change default behavior, such as variables defined in “FIRSTPRIVATE”, “PRIVATE”, and “REDUCTION”. For OpenMP compiler, some private variables need initialization and combination before or after parallel construct, like “FIRSTPRIVATE” and “REDUCTION”. Access to these data needs to be synchronized.

On *3SOC* platform, there are two levels of shared memory: local memory within Quad and shared DRAM outside Quad. The local data memory has a limited size of 64KB. Accordingly, there are four types of declarations for variables in *3SOC*, defined as:

- “_SD” - Shared DRAM, shared by all PEs.
- “_SL” - Shared Local memory of a Quad and shared by all PEs within Quad
- “_PD” - Private DRAM, allocated to one PE
- “_PL” - Private Local memory, allocated in local memory to one PE

By default, if none of these are declared, the variable is considered _PD in *3SOC*. The sample OpenMP code below illustrates data attributes:

```
1. int v1, s1[10];
2. void func() {
3.    int v2, v3, v4;
```

```
4.   #pragma omp parallel shared(v1,v2)
     firstprivate(v3) reduction(+: v4)
5.    {…}
6. }
```

If variables v1 and v2 are declared as shared, it should be defined in the global shared region. v1 is already in global shared memory, no special action needs to be taken, the only thing is to re-declare it as _SD(shared DRAM) or _SL(shared local). v2 is within the lexical context of *func()* which is a private stack area. This private stack is not accessible to other processors if it's not moved from *func()* to global shared memory during compilation. "FIRSTPRIVATE" is a special private clause that needs to be initialized before use. A global variable (v3_glb) needs to be defined and copied to the private variable (v3) before the execution of parallel construct. "REDUCTION" is another clause which needs global synchronization and initialization. A global variable is also defined in shared region and each local copy (_PL) is initialized according to reduction operation. At the end of parallel region, modification from each processor is reflected to the global variable.

For embedded systems, memory allocation is a crucial factor for performance because the local data memory is connected to a high-speed local bus. For standard OpenMP programs, data declaration in OpenMP should be treated differently for memory allocation in *3SoC*. This is a challenging task since the local memory has limited size for all CMP or equivalent DSP processors. It's not applicable to define all shared and private data structure into local memory first. We must do memory allocation dynamically in order to produce better performance.

We design to do memory allocation using "*request-and-grant*" model during compilation. At compilation, compiler will retrieve all memory allocation requests from each parallel region agreed on by some algorithm or policy-based methods. Then complier will assign different memory to different variable. For example, based on the size of data structure or frequency of referencing in the region, it will grant memory to different data structure. We also provide extensions to OpenMP to use the special hardware feature of CMP to allocate memory at runtime. (See 4.2.1 and 4.2.2)

4. Extensions to OpenMP

In a Chip Multiprocessor environment, there are several unique hardware features which are specially designed to streamline the data transfer, memory allocations, etc. Such features are important to improve the performance for parallel programming on CMP. In order to incorporate special hardware features for CMP, we extend OpenMP directives for this new parallel architecture.

4.1 OpenMP Extensions for DSE Processors

To deal with the heterogeneity of different processors within CMP, we try to incorporate DSEs into our OpenMP compiler. Unlike PE, DSE uses different programming

methodology which combines C and C-like Assembling Language (CLASM). We extend OpenMP with a set of DSE directives based on *3SOC* platform.

1. *#pragma omp parallel USING_DSE(parameters)*
   This is the main parallel region for DSEs.
2. *#pragma omp DSE_DATA_ALLOC*
   This is within DSE parallel region and used to define data allocation function.
3. *#pragma omp DSE_LOADCOMREG*
   Define data registers to be transferred to DSE
4. *#pragma omp DSE_LOADDIFFREG(i)*
   Define DSE data register with different value.
5. *#pragma omp DSE_OTHER_FUNC*
   Other user defined functions

The main parallel region is defined as *#pragma omp parallel USING_DSE (parameters)*. When the OpenMP compiler encounters this parallel region, it will switch to the corresponding DSE portion. The four parameters declared here are: *number of DSEs, number of Registers, starting DPDM number, and data register array,* such as *(8, 6, 0, dse_mem)*. “Number of DSEs” tells the compiler how many DSEs are required for parallel computation. Number of DSE data registers involved is defined in “number of Registers”. There is an array where the PE stores the data that has to be transferred into the data registers of the DSE. The PE initializes the array and calls a MTE function to transfer the data into the DSE data registers. This array is defined in “data register array.” In this example, it is *dse_mem*. The “starting DPDM number” is also required when loading data registers.

```
void main() {
        //other OpenMP parallel region
        #pragma omp parallel
        {...}
        ...
        //OpenMP parallel region for multiple DSEs
        #pragma omp parallel USING_DSE(8,6,0,dse_mem)
          {
             #pragma omp DSE_DATA_ALLOC
             {
                <initialization functions>
             }
             #pragma omp DSE_LOADCOMREG
             {
                <define data registers to be transferred to DSE>
             }
             #pragma omp DSE_LOADDIFFREG(i)
             {
                <define DSE data registers with different value>
             }
             #pragma omp DSE_OTHER_FUNC
             {
                <other user defined functions>
             }
             //main program loaded and started by PE
             #pragma omp DSE_MAIN
             {
                <order of executing main code>
             }
          }
}
```

Fig. 6. Sample OpenMP program using DSE extensions

Instead of writing *3SoC* parallel program using multiple DSEs in Fig. 4, user can write equivalent OpenMP program using DSE extensions shown in Fig. 6. For our OpenMP compiler, the code generation is guided by the parameters defined in *"parallel USING_DSE"* construct. The compiler will generate initialization and environment setup like *dse_lib_init(&LocalState),dse_allo(0),* DSE startup and wait call *dse_start(), dse_wait(),* and termination library call *dse_lib_terminate().* So users will not do any explicit DSE controls, like startup DSE *dse_start().* The DSE parallel region can also co-exist with standard OpenMP parallel regions that will be converted to parallel regions using multiple PEs.

The benefit of using OpenMP extensions is that it helps to do high-level abstraction of parallel programs, and allows the compiler to insert initialization code and data environment setup when necessary. Users are not required to focus on how to declare system data structures for DSE, and how PE actually controls multiple DSEs by complicated system calls. This will hide DSE implementation details from the programmer and greatly improve the code efficiency for parallel applications. Performance evaluation between different numbers of DSEs based on our OpenMP compiler will be given in section 6.

4.2 OpenMP Extensions for Optimization on SOCs

For SOCs or other embedded systems, memory allocation is critical to the overall performance of parallel applications. Due to the limited size of on-chip caches or memories, techniques have to be developed to improve the overall performance of SOCs. [7, 8].

4.2.1 Using MTE Transfer Engine.

Given the availability of local memory, programs will achieve better performance in local memory than in DRAM. But data locality is not guaranteed due to small on-chip caches or memories. One approach is to allocate data in DRAM first, then move data from DRAM to local memory at run-time. Thus, all the computation is done in local memory instead of slow DRAM. In *3SOC*, a user can invoke one PE to move data between the local memory and DRAM at run-time.

3SOC also provides a better solution for data transfer using MTE. MTE processor is a specially designed memory transfer engine that runs in parallel with all other processors. It transfers data between local data memory and DRAM in the background. We also incorporate MTE extensions to our OpenMP compiler.

1. *#pragma omp MTE_INIT(buffer size, data structure, data slice)*
   MTE_INIT initializes a local buffer for designated data structure.
2. *#pragma omp MTE_MOVE(count, direction)*
   MTE_MOVE will perform actual data movement by MTE engine.

MTE_INIT initializes a local buffer for *data structure* with specified *buffer size. MTE_MOVE* will perform actual data movement by MTE engine. Data size of equaling *count*slice* will be moved with respect to the *direction* (from local to DRAM or DRAM to local). Within a parallel region, a user can control data movement between

local memory and SDRAM before or after the computation. Accordingly, the MTE firmware needs to be loaded and initiated by PE0 at the beginning of the program. A number of MTE library calls will be generated and inserted by the OpenMP compiler automatically during compilation.

With the help of these extension, user can write OpenMP parallel program which controls actual data movement dynamically at run-time. The results show significant performance speedup using the MTE to do data transfers, especially when the size of target data structure is large. Performance evaluation of using the MTE versus using the PE to do memory transfer is given in section 6.

5. Implementation

In this section, we will discuss our implementation of the OpenMP compiler/translator for *3SoC*. However, this paper is not focused on implementation details. To implement an OpenMP compiler for *3SOC*, there are four major steps.

1. **Parallel regions**: Each parallel region in the OpenMP program will be assigned a unique identifying function number. The code inside the parallel region is moved from its original place into a new function context. The parallel construct code will be replaced by code of PE0's allocating multiple PEs or DSEs, setting up environment, starting all processors, assigning workload to each processor, and waiting for all other processors to finish.
2. **Data range**: Through analysis of all the data attributes in the OpenMP data environment clause, i.e., "SHARED", "PRIVATE", "FIRSTPRIVATE", "THREADPRIVATE", "REDUCTION", compiler determines the data range for separate functions and assign memory allocation like "_SL", or "_SD" in *3SOC*. Related global variable replication such as "REDUCTION" is also declared and implemented. Similar approach has been taken in Section 3.3.
3. **Work sharing constructs**: These are the most important constructs for OpenMP, referred as *for*-loop directive, *sections* directive and *single* directive. Based on the number of processors declared at the beginning of the *3SOC* program, each processor will be assigned its portion of work distinguished by processor ID. During runtime, each processor will execute its own slice of work within designated functions. For example, for the "sections" construct, each sub-section defined in *#pragma omp section* will be assigned a distinct processor ID, and run in parallel by different processors.
4. **Synchronization:** There are a number of explicit or implicit synchronization points for OpenMP constructs, i.e., *critical,* or *parallel* construct. Correspondingly, these constructs are treated by allocating a number of hardware semaphores in *3SOC*. Allocation is achieved statically or dynamically.

The first version of our compiler can take standard OpenMP programs. With our extension, user can also write OpenMP programs for advanced features of CMP, like using multiple DSEs.

6. Performance Evaluation

Our performance evaluation is based on *3SOC* architecture; the execution environment is the *3SOC* cycle accurate simulator, *Inspector* (version 3.2.042) and the *3SOC* processor. Although we have verified the programs on the real hardware, we present results on the simulator as it provides detailed profiling information. We present results of our preliminary evaluation.

To evaluate our OpenMP compiler for *3SOC*, we take parallel applications written in OpenMP and compare the performance on multiple processors under different optimization techniques: without optimization, using data locality (matrices in local memory), using the MTE for data transfer, and using the PE for data transfer. We also show the compiler overhead by comparing the result with hand-written code in *3SOC*.

Figure 7 shows the results of matrix multiplication using multiple PEs. The speedup is against sequential code running on single processor (one PE). Figure 8 is the result for LU decomposition using multiple PEs against one PE. By analysis of both charts, we conclude the following:

1. **Local memory vs DRAM:** As expected, memory access latencies have significant effect on performance. When the size of the data structure (matrix size) increases, speedup by allocation of data in local memory is obvious. For 64*64 matrix LU decomposition, the speedup is 4.12 in local memory vs 3.33 in DRAM.
2. **Using the MTE vs data in DRAM only:** As discussed in Section 5, we can deploy the MTE data transfer engine to move data from DRAM to local memory at runtime, or we can leave the data in DRAM only and never transferred to local during execution. For small size matrices below 32*32, the MTE transfer has no benefit; in fact, it downgrades the performance in both examples. The reason is that the MTE environment setup and library calls need extra cycles. For larger-size matrices, it shows speedup compared to data in DRAM only. For 64*64 matrix multiplication, the speedup is 4.7 vs 3.9. Actually 64*64 using MTE engine is only a 3.2% degrade compared to storing data entirely in the local memory. Therefore, moving data using MTE will greatly improve performance for large data structure.
3. **Using the MTE vs using the PE:** We observed scalable speedup by using the MTE over the PE to transfer memory. The extra cycles used in MTE movement do not grow much as the matrix size increases. For large data set movements, the MTE will achieve better performance than the PE.
4. **Using OpenMP compiler vs hand-written code:** The overhead of using the OpenMP compiler is addressed here. Since the compiler uses a fixed method to distribute computation, combined with extra code inserted to the program, it is not as good as manual parallel programming. In addition, some algorithms used in parallel programming cannot be represented in OpenMP. The overhead for OpenMP compiler is application dependent. Here we only compare the overhead of the same algorithm deployed by both the OpenMP compiler and hand-written code. It shows overhead is within 5% for both examples.

Figure 9 shows the result of matrix multiplication using multiple DSEs. The matrix size is 128*128.

1. **Scalable speedup using different number of DSEs:** 4 DSEs achieve 3.9 speedup over 1 DSE for the same program, and 32 DSEs obtain 24.5 speedup over 1 DSE. It shows that *3SOC* architecture is suitable for large intensive computation over multiple processors on one chip and performance is scalable.

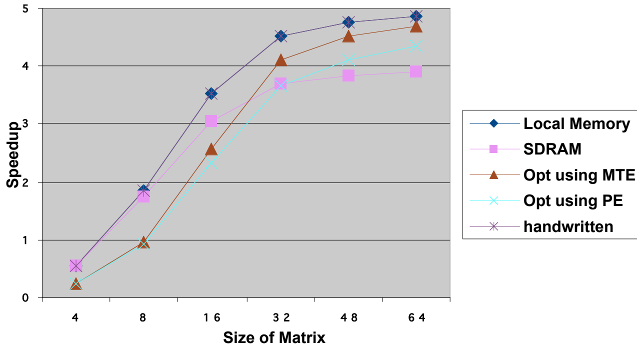

Fig. 7. Matrix Multiplication using four PEs

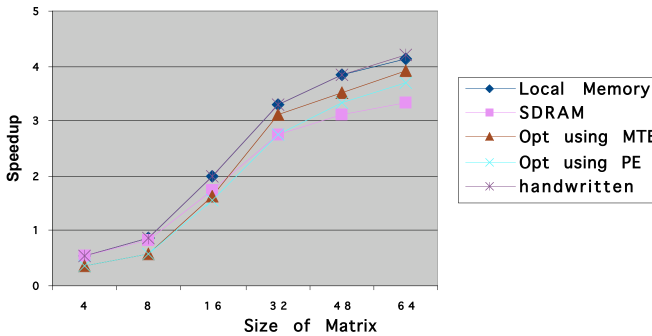

Fig. 8. LU decomposition using four PEs

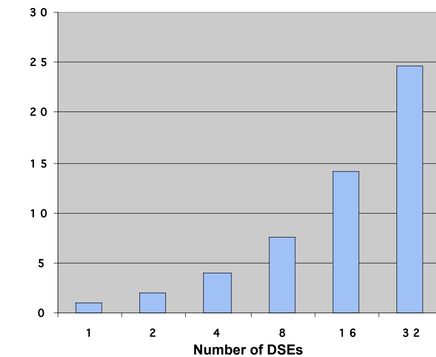

Fig.9. Matrix Multiplication using DSEs

7. Conclusion

In this paper, we present a practical OpenMP compiler for System on Chips, especially targeting *3SOC*. We also provide extensions to OpenMP to incorporate special architectural features. The OpenMP compiler hides the implementation details from the programmer, thus improving the overall efficiency of parallel programming for System on Chips architecture or embedded systems. Results show that these extensions are indispensable for performance improvement of OpenMP programs on *3SoC*, and such compilers would reduce the burden for programming SOCs. We plan to evaluate the translator/compiler on other large DSP applications and the new OpenMP benchmarks [9].

Acknowledgement

We thank the reviewers for their contributive comments on a draft of this paper. We also acknowledge the support of the Institute for Manufacturing Research at Wayne State University.

References

[1] OpenMP Architecture Review Board, OpenMP C and C++ Application Program Interface, Version 2.0, March, 2002. *http://www.openmp.org*
[2] OpenMP Architecture Review Board, OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering, Vol. 5, No. 1,* January/March 1998, http://www.openmp.org
[3] *3SOC* Documentation – *3SOC* 2003 Hardware Architecture, *Cradle Technologies, Inc.* March. 2002
[4] *3SOC* Programmer's Guide, *Cradle Technologies, Inc.,* Mar. 2002, http://www.cradle.com
[5] Christian Brunschen, Mats Brorsson, OdinMP/CCp – A portable implementation of OpenMP for C, *Lund Universtiy, Sweden, 1999*
[6] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2001
[7] Chong-Liang Ooi, Seon Wook Kim, and Il Park. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor, *Proceedings of the 15th international conference on Supercomputing* June 2001
[8] Non-Uniform Control Structures for C/C++ Explicit Parallelism, Joe Throop, Kuck & Associates, USA Poster Session at ISCOPE'98
[9] SPEC OMP Benchmark Suite, http://www.specbench.org/hpg/omp2001