Implementing OpenMP on a high performance embedded multicore MPSoC

Barbara Chapman, Lei Huang
University of Houston
Houston, TX, USA
Email: {chapman,leihuang}@cs.uh.edu

Eric Biscondi, Eric Stotzer, Ashish Shrivastava and Alan Gatherer
Texas Instruments
Houston, TX, USA
Email: {eric-biscondi,estotzer,ashu,gatherer}@ti.com

Abstract

In this paper we discuss our initial experiences adapting OpenMP to enable it to serve as a programming model for high performance embedded systems. A high-level programming model such as OpenMP has the potential to increase programmer productivity, reducing the design/development costs and time to market for such systems. However, OpenMP needs to be extended if it is to meet the needs of embedded application developers, who require the ability to express multiple levels of parallelism, real-time and resource constraints, and to provide additional information in support of optimization. It must also be capable of supporting the mapping of different software tasks, or components, to the devices configured in a given architecture.

1. Introduction

The rapid penetration of multicore and simultaneous multithreading hardware technology in the marketplace is changing the nature of application development in all areas of computing. The use of multicore technology, which offers more compute power with lower energy consumption, will increase the design space and significantly increase the performance of Multiprocessor Systems-on-Chip (MPSoCs) that are targeted at embedded applications. Software for embedded systems will become more complex, as multicore hardware will enable more functions to be implemented on the same device. This complexity will make the production of software the critical path in embedded systems development and make hardware/software designing, partitioning, scheduling, correctness verification, quality control and quality assurance more difficult.

In this paper, we discuss our initial work to adapt an existing shared memory programming interface, OpenMP, to enable the specification of high performance embedded applications. We are considering new language features that allow non-functional characteristics or constraints, such as deadlines, priorities, power constraints, etc., to be specified as well as notation to enable a high-level description of the components of an application. We are also considering how code expressed in OpenMP may be translated to run on different kinds of embedded devices. Since we also require the explicit specification of interfaces and data transfers between components, the result will be a hybrid distributed/shared programming model that can be used to model both the individual tasks within an embedded application as well as their interactions. Our implementation is based upon OpenUH, our robust Open64-based compiler infrastructure, which will be connected with a native code generator to permit us to evaluate them on a real-world system.

This paper is organized as follows. In Section 2, we give a brief overview of the state of the art in programming embedded systems. Then we describe OpenMP, its implementation, and our compiler. Section 4 gives our current ideas on the enhancements needed to this API. We next describe the current state of our implementation, followed by early results. Finally, our conclusions are discussed in Section 7.

2. Embedded Systems Design

2.1. Hardware

Embedded system designers already rely extensively on architectural parallelism to accomplish their goals. MPSoCs are composed of multiple heterogeneous processing elements or cores, such as RISC processors, digital signal processors (DSP), application specific instruction processors (ASIP), field programmable gate arrays (FPGA), stream processors, SIMD processors, accelerators, etc. In addition to the heterogeneity of their cores, an MPSoC's memory system can be heterogeneous where blocks of memory are accessible to a subset of the processors. Texas Instruments DaVinci [1] and OMAP 3 [2] are RISC + DSP configurations. Texas Instruments TNETV3020 [3] and TMS320TCI6488 [4], Along with an increase in the number and heterogeneity of cores, MPSoCs with multiple subsystems that are able to exploit both asymmetric chip multiprocessing (AMP) and symmetric chip multiprocessing (SMP) will begin to appear in the near future [5].

This work is supported by the National Science Foundation, under contracts CCF-0833201 and by Texas Instruments.

2.2. Embedded Applications

Embedded systems applications can be broadly classified as event-driven or compute and data intensive. In event-driven applications, such as automotive embedded systems, the behavior is characterized by a defined sequence of responses to a given incoming event from the environment. While the actions performed may not be compute or data intensive, they tend to have strict real time constraints. In contrast, compute and data intensive applications, such as wireless infrastructure, telecommunication systems, biomedical devices and multimedia processing, must be designed to handle and perform computations on large streams of data. As the computational complexity increase for such systems, better design methodologies are needed to find an optimal software/hardware configuration while meeting cost, performance, and power constraints. Because growth in complexity is perhaps the most significant trend in wireless and multimedia applications, more and more functionality is being implemented in software to help shorten the design cycle. Furthermore, the use of streamlined Application Specific Instruction Set Processors (ASIPs) are often employed instead of general purpose microprocessors to help limit costs and improve power efficiency [6].

2.3. Programming Embedded Systems

The approach taken today to create embedded systems is not conducive to productivity. Embedded applications are typically developed in a programming language that is close to the hardware, such as C or assembly, in order to meet strict performance requirements. It is considered impractical to program high performance embedded applications using Java or C# due to resource constraints of the device. The dominant trend in considering enhancements for MPSoCs is to adapt existing programming languages to fit the strict requirements of applications in the embedded domain, rather than designing new parallel programming languages. There are no industry standards for programming MPSoCs, although a few have been proposed. In particular, the Multicore Association has proposed a message passing model which they refer to as Communications API (CAPI)[7], but this model is not yet completed.

In the main, loosely-coupled approaches for designing and developing software are taken, where the codes for different devices are programmed separately, using different programming languages and runtime environments and system-specific mechanisms are used to enable the necessary interactions. For example, GPU-based programming models [8], [9] are dedicated to GPU devices and are not appropriate for general purpose CPUs. OpenCL [10] was recently introduced as a de facto industry standard for GPU programming. Several vendors have defined their own languages or have extended the C language to address the parallelism on their custom devices (e.g. Clearspeed's Cn and Intel's Ct).

Data and compute intensive embedded applications can often be described using the stream programming model. Characteristics include large streams of data with high data rates to and from data sources and among processing elements. Examples include signal and image processing of voice and video data, data compression and encryption, cell phones, cell phone base stations, financial modeling and HDTV editing consoles. A streaming program typically has two levels. The stream level controls the order of kernel execution and transfer of data streams for each kernel. The kernel level represents the actual computation for each stream element. The appeal of this model is apparent from a plethora of related research [11], [12].

3. OpenMP and OpenUH

OpenMP is a widely-adopted shared memory parallel programming interface providing high level programming constructs that enable the user to easily expose an application's task and loop level parallelism in an incremental fashion. Its range of applicability was significantly extended recently by the addition of explicit tasking features [13]. OpenMP has already been proposed as the starting point for a programming model for heterogeneous systems by Intel, ClearSpeed, PGI and CAPS SA.

The idea behind OpenMP is that the user specifies the parallelization strategy for a program at a high level by annotating the program code; the implementation works out the detailed mapping of the computation to the machine. It is the user's responsibility to perform any code modifications needed prior to the insertion of OpenMP constructs. In particular, it requires that dependences that might inhibit parallelization are detected and where possible, removed from the code. The major features are directives that specify that a well-structured region of code should be executed by a team of threads, who share in the work. Such regions may be nested. Worksharing directives are provided to effect a distribution of work among the participating threads.

3.1. Implementation of OpenMP

Most OpenMP compilers translate OpenMP into multithreaded code with calls to a custom runtime library in a straightforward manner, either via outlining [14] or inlining [15]. Since many execution details are often not known in advance, much of the actual work of assigning computations must be performed dynamically. Part of the implementation complexity lies in ensuring that the presence of OpenMP constructs does not unduly impede sequential optimization in the compiler. An efficient runtime library to manage program execution is essential.

IBM provides an OpenMP compiler [16] that presents the user with a shared memory abstraction of the Cell B.E. Their work focuses on optimizing DMA transfers between the SPE's local stores. This abstraction results in reduced performance, but creates a more user-friendly abstraction. [17] implements OpenMP on an MPSoC consisting of multiple RISC and DSP cores. Language extensions are proposed to specify whether a parallel region will execute on a RISC or DSP platform, and to support DMA transfers. The implementation proposed in [18] targets details related to shared variables, reductions, and synchronization mechanisms and does not rely on an operating system. However, they do not concern themselves with real-time constraints and are platform specific.

An efficient implementation of OpenMP to express the overall computation on heterogeneous MPSoCs needs to minimize thread overhead, optimize memory usage and overlap data transfer between different devices with computation as much as possible. Architectural limitations such as memory space and power consumption need to be handled carefully. The OpenMP runtime library needs to support thread management for the different devices. It needs to be further optimized to support task management and scheduling, shared memory and fine-grained synchronization.

3.2. OpenUH

The work described here makes use of the OpenUH [15] compiler, a branch of the open source Open64 compiler suite for C, C++, and Fortran 95 developed at the University of Houston. OpenUH contains a variety of state-of-the-art analyses and transformations, sometimes at multiple levels. We have implemented OpenMP 2.5 for all 3 input languages in OpenUH, and designed and implemented novel extensions that provide greater flexibility and scalability when mapping work to many CPUs or cores [19]. OpenMP is lowered relatively early in the translation process to enable optimization of the explicitly parallel code. The output makes calls to our runtime library, which is based upon PThreads. OpenUH provides native code generation for IA-32, IA-64 and Opteron architectures. It has a source-to-source translator that translates OpenMP code into optimized portable C code with calls to the runtime library, which enables OpenMP programs to run on other platforms. Its portable, multithreading runtime library includes a built-in performance monitoring capability [20]. Moreover, OpenUH has been coupled with external performance tools [21]. It is directly and freely available via the web (http://www.cs.uh.edu/∼openuh).

4. Embedded OpenMP Programming Model

A unified abstraction to tightly couple the different devices could greatly facilitate programming on heterogeneous MPSoCs. We are not the first to consider OpenMP in this context. Intel proposes the use of OpenMP along with their EXOCHI execution support. They aim to facilitate the mapping of parallel regions in an OpenMP code to different hardware components. CAPS [22] provides an alternative approach that more closely resembles our proposed model at the implementation level. Its set of OpenMP extensions target heterogeneous shared and distributed memory systems and allow for the specification of procedures that will be executed on an accelerator. However, their features are limited in their scope. PGI [23] has provided an API based on OpenMP concepts for programming heterogeneous systems under which regions of code are marked for acceleration. Clearspeed also has a prototype implementation [24] of OpenMP extensions for their accelerators.

Researchers have proposed extensions to OpenMP to facilitate the expression of streaming applications through the use of a tasking construct and support for point-to-point synchronization between tasks [25]. Carpenter et al [26] introduce a set of OpenMP-like directives to translate a C program onto a streaming machine with the support of the compiler cost model to possibly enable code to map well to different architectures. The drawback of this approach is that it does not allow the user to easily experiment with different mapping configurations. Furthermore their work is not implemented on embedded architectures. Support of the stream programming model is also being explored as part of the ACOTES project [27], which provides syntax associated with OpenMP tasks for specifying their input and output data streams.

While an integrated shared memory programming model based on OpenMP would permit incremental application creation and hide architectural complexity from programmers, more research is necessary to improve its expressivity, scalability and portability. In particular, there is potentially a high cost associated with the provision of shared virtual memory across an MPSoC. OpenMP currently assumes homogeneous processing elements and a global address space, and runtime overheads for synchronization, task management, and memory consistency may be high. It provides no support to specify constraints, and has no error handling functions. These shortcomings must be addressed in order to provide a programming model that is truly useful for real world embedded applications..

We consider the addition of information to parallel regions, worksharing directives and tasks to support embedded programming. We plan to introduce streams into OpenMP, which has the advantage of increasing parallelism via pipelining execution and locality. We are experimenting with additional "in" and "out" clauses to OpenMP worksharing and task constructs to specify the streams. We plan to provide syntax to define and connect streams, in the form of a clause to connect OpenMP work sharing and task constructs, and allow users to specify the stream size. The stream size and data transfer granularity has a big

```
#pragma omp parallel
{
#pragma omp master
 {  // create M tasks
//define priority
#pragma omp priority P
//create a task and define its priority, in
//and out stream sets
#pragma omp task in(s1) out(s2)  priority(P)
   converter(s1, s2);
#pragma omp task in(s2) out(s3) priority(P)
  compression(s2,s3);
 }
}
```

```
converter(s1, s2)
{
 #pragma omp parallel
 {
#pragma omp single onthreads(team1)
   DoSthElse()
 #pragma omp for onthreads(team2)
 For(int i=0;i<N;i++)
   s2[i] = process(s1[i]);
 }
}
```

Figure 1. An example of using OpenMP constructus to define multiple components, nested parallelism and priority of tasks

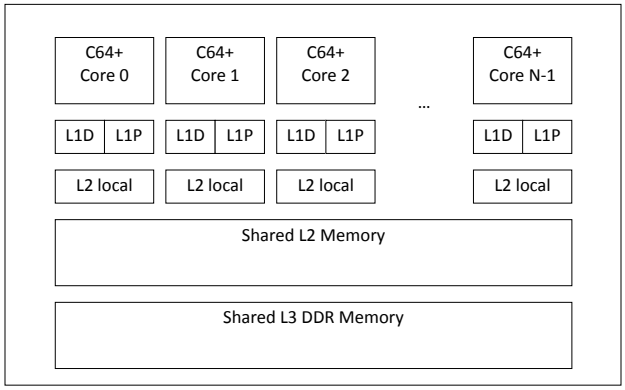

Figure 2. N-core multicore C64+ MPSoC block diagram

impact on performance. Figure 1 is a simple example of signal processing and compression that illustrates our initial thoughts on how to extend OpenMP to define tasks with different priorities, and to define in and out sets of a task. These in and out sets can be used to generate a pipelined execution of these tasks as streams. In the example, a stream of data is processed by a converter and its output is used by the other task, which compresses the data. The two tasks may be executed on different hardware components of an MPSoC.

We will examine how best to introduce real-time constraints, and determine scheduling techniques that optimize the execution of the different granularities of parallelism. We also plan to investigate the role of subteams [19] and OpenMP tasks in this context. The subteam concept organizes OpenMP threads into subsets of teams and allows worksharing constructs to be assigned to each subteam of threads. The concept may provide modularity to OpenMP, increase its parallelism and improve data locality. Since the developer will often desire the ability to specify the hardware on which a piece of code should execute, we will provide syntax to permit this for parallel regions, and worksharing directives. We plan to introduce a "target" clause to worksharing directives to specify the targeted hardware components that the worksharing enclosed code is executed on. For example. "omp task target(DSP)" indicates that a compiler will generate DSP code for the task.

Figure 1 shows how the subteam concept (onthreads clause) can be used to schedule work to different subteams of threads. These tasks/components can be dispatched to different hardware components and further parallelization specified within each individual task. The specification of priority/realtime constraints can help in the scheduling of tasks at runtime. Data attribute extensions may help the compilermake good decisions on where to store data in embedded system memory. We further intend to explore extensions that will let the developer specify characteristics of the code to the compiler, so that the compiler can support the mapping of software to hardware.

5. Case Study: Implementing OpenMP on a Multicore DSP

For this case study we provide an initial OpenMP implementation for a high performance embedded multicore based upon the Texas Instruments TMS320C64x+ processor (see Figure 2). We aim to accomplish the following:

1) Develop a predominantly shared and restricted distributed memory programming model based on OpenMP that provides both the necessary control over program execution and a high level of abstraction.
2) Implement a programming model based upon a processor widely available to the industry .
3) Develop compiler tools that can target different processor and memory configurations.
4) Leverage OpenUH and its performance tools to implement our programming model.
5) Evaluate our proposed system using a real-world embedded application.

5.1. Multicore DSP System

5.1.1. The TMS320C64x+ VLIW DSP. The TMS320C64x+ is a fully pipelined VLIW processor, which allows eight new instructions to be issued per cycle. All instructions can be optionally guarded by a static predicate. The C64x+ is a member of the Texas Instruments' C6x family (Figure 3), providing a foundation of integer instructions. It has 64 static general-purpose

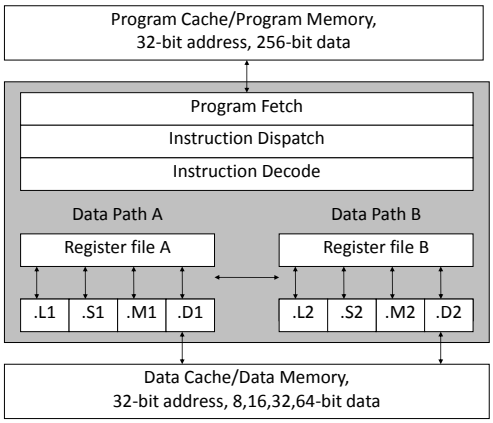

Figure 3. c64x+ processor core

registers, partitioned into two register files. A small subset of the registers may be used for predication. The TMS320C64x+ has additional instructions for packed data SIMD (single instruction multiple data) processing, variable length instructions, and a hardware loop buffer [28], [29].

The C6x targets embedded DSP (digital signal processing) applications, such as telecom and image processing. These applications spend the bulk of their time in computationally intensive loop nests which exhibit high degrees of instruction level parallelism. Software pipelining is key to extracting this parallelism and exploiting the many functional units.

5.1.2. Memory subsystem. Our multicore MPSoC model has N C64x+ DSP subsystems and a large amount of on-chip memory organized as a two-level memory system. The level-1 (L1) program and data memories on each DSP subsystem are 32KB each. This memory can be configured as mapped RAM, cache, or some combination of the two. The local level 2 (L2) memory is shared between L1 program and L1 data space and is 608K-Byte in size. Local L2 memory can also be configured as mapped RAM, cache, or some combination of the two.

All C64x+ DSP subsystems share 768 KB of L2 RAM that is cacheable by any L1D and L1P caches. Therefore, multiple copies of the same shared memory location may exist simultaneously in the L1 caches and a cache coherence protocol must be implemented. The shared L2 memory controller does not maintain coherency among the DSP subsystems, and it is the responsibility of any running program to use synchronization mechanisms and cache control operations to maintain coherent views of the memories. Coherency within a given DSP subsystem for all levels of memory (i.e. its local L1s and L2 as well its view of shared L2 and L3 DDR memory) is maintained by the hardware.

Threads executing on different DSP subsystems share data using a centralized shared memory (on chip shared L2 or external shared L3 DDR memory) or distributed memories (i.e. distributed local L2). In the former case, the application needs to explicitly manage thread synchronization and cache coherence. In the latter case, a processor can access the local L2 of another processor (with a slower access time) or a DMA controller can be used to transfer a data buffer from one local L2 to another L2. A snoop-based hardware protocol maintains L1D cache coherency with the local L2 RAM for DMA accesses, therefore there is no cache coherency operations required. Also, the DMA transfer completion event can also be used as a synchronization event between the data producer and data consumer. There is no virtual memory management (no MMU), but a memory protection mechanism protects some shared memory to be accessed/written by a non authorized processor.

5.1.3. Thread Support. Texas Instruments provides a light-weight scalable real-time operating system (RTOS), DSP/BIOS, that is designed to require minimal memory and CPU cycles [30], [31]. DSP/BIOS implements a thread priority-based preemptive multithreading model for a single processor. High priority threads preempt lower priority threads. DSP/BIOS provides for various types of interaction between threads, including blocking, communication, and synchronization. It provides support for several types of program threads (hardware interrupts, software interrupts, tasks and idle thread) with different priorities.

DSP/BIOS also handles memory management with real-time embedded application constraints. Unlike a general purpose operating system (that typically takes advantage of a MMU to implement virtual memory and dynamic memory allocation), RTOS provides speed-optimized memory allocation schemes that avoid memory fragmentation. Other DSP/BIOS modules provides an abstraction of the underlying hardware to manage the inter-tasks communication (like message queue) and synchronization (like semaphores).

5.1.4. Synchronization Primitives. There are several synchronization hardware constructs (for example a hardware semaphore, an atomic access monitors, and more) that can be used to implement synchronization primitives (like a lock, a mutex, or the implementation of a barrier).

On our multicore MPSoC model, the shared L2 memory controller includes an atomic access monitor to allow a mutual exclusion for concurrent accesses. The shared L2 memory controller supports the load-link (LL), store-link (SL), and a commit-link (CMTL) operations. The hardware mechanisms are hidden by DSP/BIOS.

5.2. Compiler

OpenUH implements OpenMP on our multicore MPSoC by performing a source to source translation from C code with OpenMP directives to explicit multithreaded C code including calls to our OpenMP runtime library (see Figure 4).

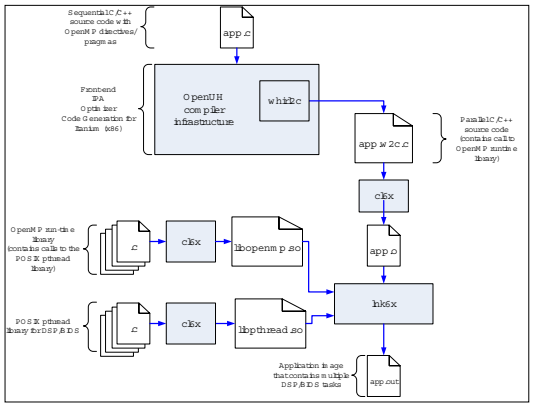

Figure 4. Code generation flow

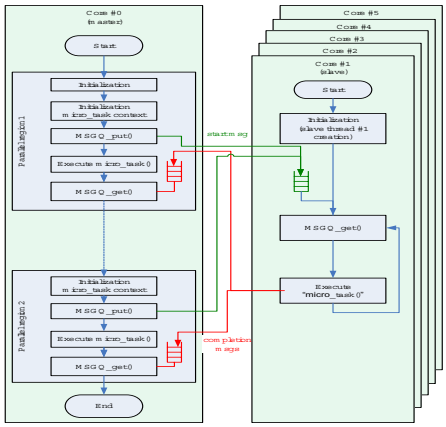

Figure 5. Fork/Join of micro-tasks

We then compile this code with the Texas Instruments C64x compiler (cl6x). We determined that a new OpenMP runtime library would have to be developed with the following considerations: the small amount of memory available, the memory layout on the device, which is distributed without cache coherence, the shared memory at L2 cache level, and the lack of an operating system that has native support for multicore.

5.3. Implementing the OpenMP Programming Model

Time spent in synchronization might significantly affect the performance of OpenMP on multi-core devices. Synchronization is used to coordinate the execution of concurrent threads and to manage accesses to the shared resources (memory, peripheral and coprocessors). While the cost of synchronization might be negligible for many coarse grained parallel applications, our OpenMP runtime library will have to be optimized for fine grained parallel applications.

OpenMP specifies a relaxed consistency shared memory model (i.e. data coherency/consistency is only required at certain points in the execution flow). Even if our multi-core DSP provides a shared memory, the coherence of the latter is not automatically maintained by the hardware. It will be the responsibility of our OpenMP runtime library to perform the appropriate cache control operations to maintain the shared memory when required. Moreover, future OpenMP runtime library implementations will have to support multi-core devices that only provide distributed memories.

5.3.1. Memory Model. One objective of our work is to study the performance of different approaches to supporting OpenMP's relaxed consistency model. OpenMP distinguishes shared data (which all the threads can access) and private data (which is only accessible by one thread). In our work, we allocate all private variables in the local L2 RAM. For the shared variables, we mainly study two approaches. In a first implementation, all shared variables are placed in the centralized shared memory (shared L2 RAM). When a thread updates a shared variable, the updated value is first stored in the L1D cache. While OpenMP provides the flush directive to force a consistent view of the shared memory, the runtime library is in charge of performing the right cache control operations for each implicit flush operation at each barrier. A second approach consists of allocating all the shared variables in the distributed memories (local L2 RAM) and using a DMA controller to maintain the coherency of these memories. The executable image is loaded in the shared L2 memory and can be accessed by all the cores taking advantage of the direct-mapped L1 program cache.

5.3.2. Work Sharing Constructs. In a very first implementation, and in order to maintain the portability of ourPthreads-based runtime library, we used a Pthreads library based on PTE (Posix Threads Embedded) [32] on top of the DSP/BIOS. The advantage of this approach is that it required minimal modification of our OpenMP runtime. The disadvantages are the fact that the implementation is limited to a single core, and the overheads associated with the POSIX and DSP/BIOS.

In a second implementation we developed a lightweight OpenMP runtime library that directly uses DSP/BIOS modules. For each parallel region, the OpenMP compiler divides the workload into multiple microtasks that are then assigned to slave threads in a team at runtime. For our lightweight implementation, we assumed a fixed allocation of the master and slave threads to the processors. As shown in Figure 5, each core runs an instance of DSP/BIOS but the first core creates and executes the master thread while slave threads are created and executed by every other core. After initialization, slave threads wait for a microtask execution

notification. The master thread assigns microtasks to slave threads by sending messages (that include the function pointer and frame pointer) through a message queue. Upon reception of the message, each slave thread executes the microtask. This mechanism basically implements the fork mechanism. Once a slave thread completes its execution of the microtask, it sends a completion message back to the master queue. Once the master completes the execution of its own microtask, it waits for all the completion messages from all the slaves (implementation of the barrier).

DSP/BIOS implements a message queue through the MSGQ module that enables the sending and receiving of variable length messages. It can be used for homogeneous or heterogeneous multicore messaging. To transport (with or without a copy) a message from one core to another across a physical link, the message queue uses a message queue transport (MQT) that can be configured for each queue. For our model MPSoC system, DSP/BIOS provides an implementation of the message queue transport that uses the shared memory (Shared Memory Message Queue Transport (SMMQT) [33]) to transport messages between different cores. The SSMQT module automatically takes care of the synchronization and cache coherence management for all the messages exchanged through the message queues by using the CRIT and SMPOOL modules from DSP/BIOS. CRIT is a module which provides an interface to synchronization hardware constructs (like atomic access monitors and hardware semaphores) that need to be used to protect critical section of code. SMPOOL is a module that manages memory in a shared memory (i.e. handling appropriately all the atomic accesses and the cache coherency protocol). The OpenMP runtime library takes advantages of these modules to correctly handle accesses to the shared variables.

In a third implementation, we used a fast synchronization mechanism using a coherent shared memory to store a vector. Each core can set/clear independently an element. Every core can concurrently query the entire vector by using single 64-bit memory access.

In the future, we plan to modify the OpenMP runtime library in order to support distributed local memories where we also map shared/private variables to different memory components in the system following the OpenMP's relaxed memory consistency model.

6. Results

A few experiments have been conducted to assess the performance of various implementation choices of the lightweight runtime library. We executed a simple OpenMP matrix multiplication program and a dot product (that implies the reduction of a temporary variable) on a C64x+ based MPSoC.

While the implementation of the synchronization (fork, join, and barrier) using the MSGQ represents an elegant and scalable approach, the associated overhead does not enable efficient execution of fine grained parallel applications. A simple optimization that consists of implementing the synchronization between cores using coherent shared buffers enables a significant reduction of the overhead.

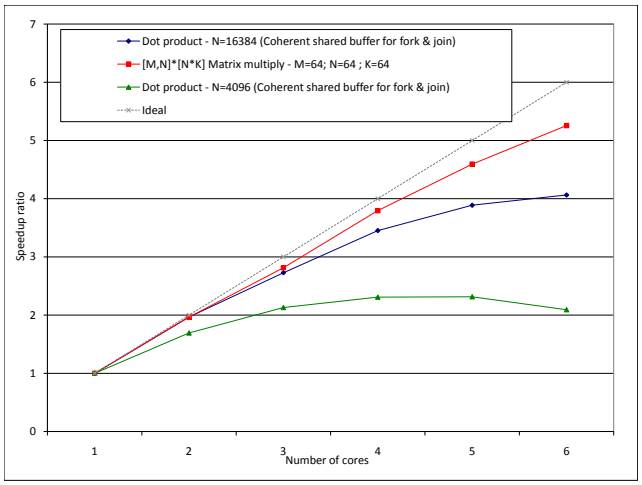

Figure 6. Efficiency for various program and workload sizes

Figure 6 shows the efficiency of the execution of the dot product (for N=4096 and N=16384) and the [M,N]*[N,K] matrix multiply. The workload of the dot product of size N=4096 is small and the overhead of OpenMP directives becomes predominant as we parallelize the program across four and more cores.

7. Conclusions

We believe that, with suitable extensions, OpenMP might become a suitable and productive programming model for embedded systems. We have begun to explore a variety of enhancements for this purpose. High performance embedded systems lack features that are commonly found in general purpose processors such as memory management units, coherent cache systems, and uniform memory access time. We are in the process of extending OpenUH to enable it to be used as an OpenMP compiler for a NUMA multicore embedded DSP system that is not cache coherent, and runs a lightweight embedded OS. This work is in its early stages. Our next steps will be to fully implement current OpenMP and further explore language extensions for embedded systems.

References

[1] T. Instruments, *TMS320DM6467 Digital Media System-on-Chip*, December 2007.

[2] ——, "Omap 3 family of multimedia applications processors," http://focus.ti.com/pdfs/wtbu/ti_omap3family.pdf, 2007.

[3] ——, "TNETV3020 carrier infrastructure platform," http://focus.ti.com/lit/ml/spat174/spat174.pdf.

[4] ——, "TMS320TCI6488 DSP platform," http://focus.ti.com/lit/ml/sprt415/sprt415.pdf.

[5] G. Martin, "Overview of the MPSoC design challenge," in *DAC '06: Proceedings of the 43rd annual conference on Design automation*. Santa Clara, CA, USA: ACM, 2006, pp. 274 – 279.

[6] P. G. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens, "Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends," in *Proceedings of the IEEE*. IEEE, 1997, pp. 419–433.

[7] T. M. Association, http://www.multicore-association.com/workgroup/ComAPI.html.

[8] GPGPU, "Gpgpu: General purpose computation using graphics hardware," http://www.gpgpu.org.

[9] NVIDIA, "Cuda: Compute unified device architecture," http://www.nvidia.com/object/cuda_home.html.

[10] "OpenCL 1.0 Specification," http://www.khronos.org/opencl/.

[11] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies, "Language and compiler design for streaming applications," *Int. J. Parallel Program.*, vol. 33, no. 2, pp. 261–278, 2005.

[12] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*. New York, NY, USA: ACM, 2004, pp. 777–786.

[13] E. Ayugade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang, "A proposal for task parallelism in openmp," in *Proceedings of the 3rd International Workshop on OpenMP*, June 2007.

[14] C. Brunschen and M. Brorsson, "OdinMP/CCp - a portable implementation of OpenMP for C," *Concurrency - Practice and Experience*, vol. 12, no. 12, pp. 1193–1203, 2000.

[15] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "OpenUH: An optimizing, portable OpenMP compiler," in *12th Workshop on Compilers for Parallel Computers*, 2006.

[16] A. Eichenberger, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing compiler for a cell processor," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 161–172.

[17] F. Liu and V. Chaudhary, "A practical OpenMP compiler for system on chips," in *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools, WOMPAT 2003*, 2003.

[18] W.-C. Jeun and S. Ha, "Effective openmp implementation and translation for multiprocessor system-on-chip without using os," in *ASP-DAC '07: Proceedings of the 2007 conference on Asia South Pacific design automation*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 44–49.

[19] B. M. Chapman, L. Huang, H. Jin, G. Jost, and B. R. de Supinski, "Toward enhancing openmp's work-sharing directives," in *Europar 2006*, 2006, pp. 645–654.

[20] V. Bui, O. Hernandez, B. Chapman, R. Kufrin, P. Gopalkrishnan, and D. Tafti, "Towards an implementation of the openmp collector api," in *PARCO*, 2007.

[21] O. Hernandez, F. Song, B. Chapman, J. Dongarra, B. Mohr, S. Moore, and F. Wolf, "Performance instrumentation and compiler optimizations for mpi/openmp applications," in *Second International Workshop on OpenMP*, 2006.

[22] C. Enterprise, "Hmpp: A hybrid multicore parallel programming platform," http://www.caps-entreprise.com/en/documentation/caps_hmpp_product_brief.pdf.

[23] "PGI Fortran & C Accelerator Compilers and Programming Model," http://www.pgroup.com/lit/pgi_whitepaper_accpre.pdf.

[24] B. Gaster and C. Bradley, "Exploiting loop-level parallelism for simd arrays using openmp," in *Proceedings of IWOMP 2007*, June, 2007.

[25] M. Gonzalez, E. Ayguade, X. Martorell, and J. Labarta, "Exploiting pipelined executions in openmp," in *International Conference on Parallel Processing*, 2003, pp. 153 – 160.

[26] P. Carpenter, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "A streaming machine description and programming model." in *SAMOS*, ser. Lecture Notes in Computer Science. Springer, 2007, pp. 107–116.

[27] "Acotes:Advanced Compiler Technologies for Embedded Streaming," http://www.hitech-projects.com/euprojects/ACOTES.

[28] E. Stotzer and E. Leiss, "Modulo scheduling for the tms320c6x vliw dsp architecture," in *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*. New York, NY, USA: ACM, 1999, pp. 28–34.

[29] T. Hahn, E. Stotzer, and M. Asal., "Compilation strategies for reducing code size on a vliw with variable length instructions," in *High Performance Embedded Architectures and Compilers (HIPEAC'08)*, 2008, pp. 147–160.

[30] T. Instruments, "Tms320 dsp/bios user's guide, spru423f," Tech. Rep., Nov. 2004.

[31] ——, "Tms320c6000 dsp/bios 5.32 application programming interface (api) reference guide, spru403o," Tech. Rep., Sept. 2007.

[32] O. ARB, "Openmp application programming interface." http://www.openmp.org/mp-documents/spec30.pdf, May, 2008.

[33] T. Instruments, "Shared memory message queue transport (mqt) and pool modules, spraa2i," Tech. Rep., Dec. 2006.