A Framework for Evaluating Programming Models for Embedded Chip Multiprocessor (CMP) Systems

Niraj Shah, Mel Tsai
CS252 Final Project
May 9, 2000
{niraj, mtsai}@ic.eecs.berkeley.edu

Abstract

The success of fully programmable embedded systems will rest on the ability to efficiently program them at a high level of abstraction. Therefore, the architectural abstraction presented to the programmer, or programming model, must provide a balance between exposing certain machine details and hiding others. For this project, we implemented a framework for quantitative evaluation of programming model features. We use this framework to evaluate an initial programming model with a network routing application kernel.

1 Introduction

Advances in silicon integration have enabled large-scale multiprocessing on a single chip. A major use of this technology will be for application specific programmable solutions. The embedded multiprocessor architectures we are targeting are significantly different than existing general-purpose multiprocessor architectures in that:

• Interprocess communication can be very cheap
• Communication architecture can be tailored to the application
• Desirable not to have a heavy OS or large library to handle communication

The MESCAL [1] platform allows us to tailor an embedded chip multiprocessor (CMP) architecture to match the system requirements. One of the goals of the MESCAL project is to develop a programming model that model defines an abstraction of the architecture that the programmer uses to describe applications to the compiler. In this project, we primarily focus on how the communication model of the architecture is exposed to the programmer. Specifically, we’ve created a framework to explore the effectiveness of programming models. We tested this framework by implementing and simulating a networking application kernel using the initial MESCAL programming model.

The rest of this paper is organized as follows: Section 2 outlines some background information. Section 3 describes the bulk of our framework, the software environment. Section 4 discusses our application driver, network routing. Sections 5, 6, and 7 describe our results, future work, and conclusions.

2 Background

Since this work was carried out within the MESCAL environment, we first describe the characteristics of MESCAL relevant to this project: the family of target architectures and the initial programming model.

2.1 Target Architectures

We are targeting the family of architectures that the MESCAL project is considering. This architectural platform is a multiprocessor distributed memory architecture, where each processing element (PE) is tailored for a particular task or set of tasks. To exploit a high degree of instruction level parallelism (ILP), each PE is an EPIC (Explicitly Parallel Instruction Computing) style processor. The degree of customization of the PE’s is based on what the IMPACT software environment can support and will be described later.

To explore the programming model for this family of architectures, we have added message-passing assembly instructions (SEND and RECV) to each PE. The semantics of the communication instructions are based on Message Passing Interface (MPI). Using the definitions outlined in the MPI standard [2], our SEND and RECV instructions are classified as:

• **Blocking**: a SEND (RECV) operation does not return until the matching RECV (SEND) has begun; and
• **Synchronous**: a SEND operation will successfully complete only if a matching RECV has started receiving the sent message.
In addition, the communication instructions are
guaranteed to preserve order (i.e. messages are
non-overtaking).

2.2 Programming Model
The programming model that we are using is initial
programming model for the MESCAL project. Our
language specification is a simplified subset of the
Message Passing Interface [2], a standard seman-
tics and programming interface for message-
passing on distributed multiprocessors. The key
aspects of our model are:

- Single Program Multiple Data (SPMD) execution
  model
- Separate address space for each process
- Each process is bound to a distinct PE
- Only synchronous blocking synchronous send
  and receive operations
- Total connectivity between PE’s
- Only one-to-one communication (does not sup-
  port broadcast, gossiping, etc.)

3 Software Environment
We built our software environment on top the
IMPACT framework [3]. We chose the IMPACT
infrastructure because it includes a retargetable
compiler and cycle accurate simulator for a wide
variety of EPIC architectures. Currently, the
framework allows modification of the following ar-
chitectural parameters: the number and type of
function units (integer ALU, floating-point ALU,
memory, and branch), cache/memory configurations
and sizes, configuration of speculation and predica-
tion, etc. In addition to traditional machine-
independent compiler optimizations (i.e. Dragon
book optimizations [4]), the compiler has extensive
VLIW support, including hyperblock and super-
block formation, if-conversion, speculative and
predicated execution, and software pipelining. The
IMPACT machine simulator is a cycle-accurate
trace simulator for performance reasons. The trace
simulator first generates C code that emulates the
input assembly program and outputs the dynamic
information needed to accurately simulate the pro-
gram (e.g. locations of loads and stores for proper
cache simulation). This C code is then compiled
and run (on the native machine) to produce trace
data. The trace, assembly code, and machine de-
scription are fed into the simulator, which simul-
ates the machine and outputs cycle-accurate simu-
lation data. Emulating the code on the native de-
velopment environment makes it possible to only
simulate (not emulate) the machine and still get
cycle-accurate data. The assembly code provides
the static information, while the trace data pro-
vides the necessary dynamic information.

3.1 Machine Description
The retargetable compiler and simulator perform
their operations based on a machine description
file, HMDES [5], which defines the number and
types of resources and the resource usage and la-
tencies (reservation table) for each instruction. To
ensure the send and receive instructions are prop-
erly scheduled by the compiler and simulated cor-
rectly, they were included in the machine descrip-
tion. Since the micro-architecture of the communi-
cation instructions has not yet been determined
with MESCAL, we simply give each PE a single
fake resource called a channel that the send and
receive instructions “use”. In addition, we tag the
communication instructions with a synchronization
flag to force the appropriate dependence edges so
the compiler does not reorder the communication
instructions. This is needed to preserve the order-
ing semantics defined in our programming model.

3.2 Compiler
An enabling part of this project is the develop-
ment of a compiler that understands our initial pro-
gramming model and generates code with the new
communication instructions we’ve added. As a re-
result, it was necessary to augment the existing
IMPACT compiler to interpret the SPMD execution
model. The compiler now copies the entire program
to each processing element. The original IMPACT
compiler is then used to optimize and generate code
for each processing element. The compiler can
declare dead code eliminate portions of code not relevant to
each processing element based on the assignment
of a global rank variable. To simplify the code gen-
eration process, we give the programmer explicit
access to the send and receive instructions via in-
trinsics. Intrinsics allow the programmer to per-
form assembly instruction selection without doing
scheduling or register allocation. Thus, we aug-
mented the IMPACT compiler to recognize these
intrinsics and generate explicit SEND and RECV in-
structions.

3.3 Trace Simulator
The bulk of this project was writing a multi-
processor simulator to gather accurate data about
the performance of our target application expressed
in our programming model. Since we built upon
IMPACT’s existing (cycle-accurate) single EPIC
processing element simulator, most of our work was
focused on writing a “wrapper” that called the sin-
gle PE simulator for each PE in our system and
writing code to simulate the communication envi-
ronment.

Since the original simulator generated functionally
correct code to obtain trace data, our simulator had
to do the same. However, we need to generate
functionally correct multiprocessing code. Since our programming model is similar to that of MPI's, we decided to use an MPI implementation to emulate our code. As a result, the C code generated from IMPACT assembly now includes MPI communication primitives. The resulting program is compiled by the MPI C compiler and executed within the MPI environment (which adds little overhead to the native development environment). The use of MPI gives us the added flexibility of being able to concurrently execute this executable on multiple machines very easily.

To simulate the communication environment, the simulator needs to know dynamic information about send and receive instructions (namely, the source and destination PE). To accomplish this, we added a probe to the generated code that prints the source PE for a SEND instruction and the destination PE for a RECV instruction. When simulating a program, the machine simulator reads this information whenever it encounters a communication instruction. The communication infrastructure is simulated by a queue that stores outstanding communication requests. Whenever a SEND or RECV instruction reaches the execute stage, the simulator adds it to the communication queue. Every cycle, this queue is examined and all pairs of matching instructions are removed from the queue and allowed to pass to the next stage of the pipeline of their respective PE's. Figure 1 shows a flowchart of the trace simulator.

All time spent blocking is recorded and output when the simulation is complete. This allows us to examine the quality of the implementation of the application in the given programming model. We must then determine whether the application needs to be rewritten to better utilize resources or that the programming model is insufficient to efficiently map express this application to the architecture.

4 Network Routing

To explore the effectiveness of our programming model, we implemented an application using our model and evaluated various aspects of our implementation. Since this project is carried out in the MESCAL environment, we chose to use their application driver — network routing. We implemented a network routing kernel with our initial programming model and evaluated it within our framework.

We chose to implement an adaptation of the MIT Click Modular Router [6], an easy to use routing package that is easily extensible using software "elements" with a common interface between them. Since the Click implementation was written in C++ and the IMPACT compiler only compiles ANSI C, we used a modified version of the software called "CRACK" (Click Rapidly Adapted to C Kode).
appropriate output port. Figure 3 shows the configuration of the routing kernel we used.

The InfiniteSource element sends random packets to CheckIPHeader, which analyzes packets’ header information and drops non-conforming packets. GetIPAddress looks for the destination IP address embedded within the header and LookupIPRoute determines which port the packet should be routed to.

![Figure 3. Routing Kernel](image)

The serial implementation was broken up into separate processes to simulate the parallelized version of the application with our trace simulator. Our initial test involved mapping each CRACK element to a separate process. Thus, we had five separate processes: InfiniteSource, CheckIPHeader, GetIPAddress, LookupIPRoute, and Discard. The Discard process simply deletes all packets it receives and is used to model the output ports of the router (i.e. it would not exist on a real implementation of a router). At first glance, this appeared to be a naive splitting of the application, however, each process performs a similar number of computations on each packet so the load balancing between processing elements is quite even.

5 Results

The results of this project can be interpreted in many ways. We can evaluate our framework by a number of metrics: the usefulness and expressability of our initial programming model, the speed and extensibility of our software environment, and the feedback our framework provided about the implementation of our target application.

5.1 Programming Model

Based on the feedback of our framework, we found the programming model quite useful for describing parallel applications. However, we felt the lack types of communication instructions other than synchronous blocking to be too restrictive. The programming model should allow the programmer to improve efficiency by overlapping communication and computation. In addition, binding each process to a distinct PE does not allow the programmer enough flexibility to efficiently use resources. Though we did need to broadcast some data from the master PE to all other PE’s, the lack of broadcasting support was not a restriction. We emulated the broadcast by performing multiple SEND operations.

5.2 Software Environment

Our framework for evaluating programming models has proven to be quite useful for the chosen programming model and application. In addition, it can be extended to handle new programming model features. Despite this extensibility, we assume total connectivity between PE’s, as MPI does. Since the communication architecture may be customized for the application, this assumption will not always be true. Based on the characteristics of the MESCAL project’s target architectures, we will revisit the assumptions made in the compiler and simulator.

The speed of the trace simulator is rather limiting; it averages only 100 system cycles per second (on a 200 MHz Pentium Pro running Linux). Aside from running the simulator on a faster machine with more memory, the poor performance can be improved greatly by implementing sampling on the multiprocessor simulator. Currently, the entire machine is simulated every clock cycle. By contrast, most simulators only simulate the entire machine during selected periods of the entire program execution and report detailed statistics based on those samples. Though this technique slightly compromises accuracy, it enables timely simulation of long running applications. The combination of a better host and sampling will result in an estimated 100x speed improvement.

5.3 Application Implementation

The results of our parallel implementation of CRACK are moderate compared to a serialized router. In [7], De Bernardinis and Weber show the results of a serial implementation of CRACK on a single PE (which is very similar to each of our PE’s). Compared to those results, we achieve about a 2.5x speedup, but use 4x the area (one PE for each process, except for Discard). The serial implementation requires that each packet be completely processed before it can start processing the next one, while our implementation can process up to four packets at once in a pipelined fashion. Each packet required (on average) about 1800 machine cycles to traverse through the parallelized router, including the communication overhead between PE’s. Though only blocking communication was used, its overhead was small, but not negligible.
compared to the computation performed on each of the PE’s.

6 Future Work
Since this project was carried out within the larger environment of the relatively young MESCAL effort, there are many future directions to this work. We plan to add non-blocking communication primitives to the programming model to allow the programmer to overlap communication and computation. We also plan to allow the programmer to bind multiple processes to a single PE to better utilize resources. In addition, the simulator must be changed to support these new features. The speed of the simulator needs to be improved greatly to enable simulation of larger programs. This will likely entail supporting sampling with communication instructions. Due to the disparate nature of the IMPACT package, extending the framework to handle new programming model features is rather painful. We are currently discussing ways to ease this process.

7 Conclusions
We have created a useful framework that enables quantitative analysis of programming language features. Though this framework is nowhere near complete, it will be a key element for designing and evaluating a programming model for embedded CMP’s.

8 Acknowledgements
We would like to thank the members of the MESCAL project for their input. In particular, thanks to Ashok Sudarsanam for his help on the compiler portion of the framework. Thanks to Scott Weber, Fernando De Bernardinis, and Michael Shilman for their work on the CRACK implementation.

9 References