AutoPilot: A Message-Passing Parallel Programming Library for the IMAPCAR2

by

Benjamin Kelly

A Thesis
Presented to
The University of Guelph

In partial fulfilment of requirements
for the degree of
Master of Science
in
Computer Science

Guelph, Ontario, Canada
© Benjamin Kelly, February, 2013
ABSTRACT

AutoPilot: A Message-Passing Parallel Programming Library for the IMAPCAR2

Benjamin Kelly
University of Guelph, 2013

Advisor:
Dr. William Gardner

The IMAPCAR2 from Renesas Electronics is an embedded realtime image processor, combining a single core with a 128-way SIMD array. At runtime, sections of the SIMD array can be retasked as additional CPU cores, interconnected via a message passing ring. Using these cores effectively, however, is made difficult by the low-level nature of the message passing API and the lack of cache coherency between processors. Developing and debugging software for this platform is a difficult task.

The AutoPilot library addresses this by providing a high-level message-oriented parallel programming model for the IMAPCAR2. AutoPilot’s API is closely based on that of Pilot, a wrapper around the Message Passing Interface (MPI) for cluster computing. By reimplementing the Pilot API for the IMAPCAR2, AutoPilot shows that its processes-and-channels architecture is a viable choice for parallel programming on cache-incoherent multicore architectures. At the same time, it provides a simpler API for programmers, with builtin safety checks that eliminate some common sources of errors.
Dedication

To my father, Tom Kelly,

who taught me many things,

including how much joy there was in programming.
Acknowledgments

I would like to thank the many people who helped me with this project:

• Dr. William Gardner, for supervising and advising me through the research process, proofreading and editing numerous revisions of this thesis, and kicking me back into motion when I lost momentum.

• Dr. Gary Grewal, for further advice and thesis editing.

• The examination committee, Dr. Blair Nonnecke and Dr. Fei Song, and for their suggestions, improvements, and insightful questioning.

• My fellow research group members, Natalie Girard and Alicia Guntie, for helping debug my presentations.

• John Carter for his work on the original Pilot library.

• Dr. Shawki Areibi, for getting me interested in reconfigurable hardware and helping me find my area of application in undergrad.

• The Renesas Electronics IMAP team, especially Shorin Kyo, for the gift of an IMAPCAR2 test board and SDK, and unstinting assistance in learning to use it.

• The Nightstar crew, especially #code regulars Reiver, Orthia, Vornicus, Derakon, and McMartin, for engaging technical discussion and general grounding.

• My family, for many things, but most recently for helping me move when I was in the final stages of editing.

• And my wife, Ashley, for loving and supporting me through the entire process no matter how stressful and crazy it got.
# Table of Contents

Dedication iii

Acknowledgements iv

Table of Contents v

List of Tables ix

List of Figures x

1 Introduction 1
   1.1 Motivation ................................................................. 1
   1.2 Objectives .................................................................. 3
      1.2.1 Thesis Statement ...................................................... 4
   1.3 Thesis Outline ............................................................... 4

2 Background and Related Work 6
   2.1 Pilot ............................................................................ 7
      2.1.1 Basic Concepts ......................................................... 7
      2.1.2 Program Structure ...................................................... 8
      2.1.3 Communication ......................................................... 10
      2.1.4 Error Detection ........................................................ 12
      2.1.5 Example Program ...................................................... 12
   2.2 The IMAP Architecture .................................................. 16
      2.2.1 IMAP-VISION .......................................................... 17
      2.2.2 IMAP-CE ................................................................. 18
      2.2.3 IMAPCAR ............................................................... 19
      2.2.4 IMAPCAR2 and the XC Core ................................. 20
   2.3 1-Dimensional C ............................................................ 26
      2.3.1 New Declarations ..................................................... 27
      2.3.2 New Operators ........................................................ 28
      2.3.3 New Keywords ......................................................... 29
   2.4 The IMAPCAR2 SDK ...................................................... 30
      2.4.1 lib1dc ................................................................. 32
   2.5 Related Work ............................................................... 33
      2.5.1 Hardware ............................................................... 33
C Benchmark Code
C.1 Mode Switching ................................................. 148
C.2 Message Passing .................................................. 150
C.3 Broadcast Messages ............................................. 152
C.4 Large Buffers ..................................................... 154

D N-Queens Code
D.1 Sequential Version ............................................. 158
D.2 SIMD Version ..................................................... 160
D.3 MIMD Version ..................................................... 163
D.4 AutoPilot Versions .............................................. 167
   D.4.1 Round-Robin .............................................. 167
   D.4.2 Non-Round-Robin ....................................... 172
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Pilot format string codes</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Format strings in AutoPilot</td>
<td>65</td>
</tr>
<tr>
<td>5.1</td>
<td>Library size comparison</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>Object File Contents</td>
<td>90</td>
</tr>
<tr>
<td>5.3</td>
<td>Profiling results for simple message passing</td>
<td>95</td>
</tr>
<tr>
<td>5.4</td>
<td>N-Queens Size Comparison</td>
<td>104</td>
</tr>
<tr>
<td>5.5</td>
<td>Runtime details for 8 Queens</td>
<td>107</td>
</tr>
<tr>
<td>5.6</td>
<td>Runtime details for 10 Queens</td>
<td>109</td>
</tr>
<tr>
<td>D.1</td>
<td>N-Queens Source Size</td>
<td>157</td>
</tr>
</tbody>
</table>
List of Figures

2.1 A simple Pilot program .................................................. 13
2.2 The communication graph for a simple Pilot program ............ 14
2.3 IMAP-VISION board configuration [16] ................................ 17
2.4 IMAPCAR2 Reconfigurable Tile Details [29] ......................... 21
2.5 IMAPCAR2 Memory Map [4] .............................................. 23
2.6 IMAPCAR2 MP Mode DMEM Map [4] ................................ 24
2.7 Execution time of a white-line detection algorithm [42] ........... 26
3.1 Direct cache manipulation to update a global counter .............. 51
3.2 Using outside to avoid the cache ......................................... 52
3.3 Region of interest access, with and without ix_roi_* functions [42] .. 54
4.1 Sequence Diagram: Read/Write (Reader First) ....................... 77
4.2 Sequence Diagram: Read/Write (Writer First) ....................... 78
5.1 Performance: Mode Switching ............................................ 93
5.2 Performance: One-To-One Messages ................................... 94
5.3 Performance: Broadcast Messages .................................... 97
5.4 Performance: Buffers ....................................................... 99
5.5 Code comparison: initialization ........................................ 100
5.6 Code comparison: read/write ............................................ 101
5.7 Speedup relative to sequential 8 Queens implementation ........... 106
5.8 Speedup relative to sequential 10 Queens implementation ........... 110
6.1 Proposed API for distribute operations ............................... 118
Chapter 1

Introduction

1.1 Motivation

Realtime image processing is a requirement of machine vision, and is especially important for robots and intelligent vehicles (“smart cars”).\(^1\) Such image processing depends heavily on data-parallel, whole-image operations that are easily implemented on Single Instruction, Multiple Data (SIMD) architectures. Increasingly, however, it also requires operations that are not amenable to SIMD designs — non-uniform data-parallel operations, such as analysis of multiple regions of interest that may vary greatly in size and content, or true task-parallel operations such as pipelined analysis algorithms [42]. Parallelization of these algorithms requires some form of Multiple Instruction, Multiple Data (MIMD) support. Specialized platforms for realtime image processing address this in a variety of ways, such as focussing entirely on one or the other, or providing hardware for both on the same chip (at the cost of increased complexity and power requirements). Section 2.5 discusses some of these platforms.

The IMAPCAR2, and the XC Core architecture it is based on, is a recent development from a research team at Renesas Electronics. Based on the Integrated Memory Array

\(^1\)This automotive application area gave rise to the project code name “AutoPilot.” See Subsection 2.2.3 for further explanation.
Processor (IMAP) architecture, where an array of SIMD processing elements use directly connected memory as a frame buffer, it addresses this issue by adding reconfiguration capability, letting it repurpose sections of the SIMD array as additional MIMD cores at runtime. However, the lack of cache coherency hardware and the nature of the available application programming interfaces (or APIs) make this difficult to use safely and debug effectively.

The IMAPCAR2 does provide caches for instructions and data, thereby cutting access times by a factor of four or more relative to main memory accesses. On commodity multicore chips, processor-specific caches are synchronized by an elaborate cache coherency mechanism, which allows the parallel programmer to largely ignore the caches. However, this technique starts to bog down as the core count rises, and eventually a “coherency wall” is reached where the overhead of keeping so many caches in sync reduces the value of adding cores [28].

The IMAPCAR2 designers decided to forgo cache coherency, which, on the one hand, allowed them to use the chip "real estate" that the coherency hardware would have consumed to fabricate more cores, and, on the other hand, avoided a feature sure to be overburdened by its 32 processing units.

In practice, this is not a problem, as long as programmers avoid relying on shared memory (and thus implicitly on cache coherence across cores). Unfortunately, the only “high-level” API currently available is the vendor’s implementation of pthreads, the POSIX threading library. Designed for single-core and cache-coherent multicore general purpose computers, pthreads encourages, and tacitly assumes, the use of shared memory between threads. While there are workarounds for this (which the IMAPCAR2 documentation details), failure to use them effectively can result in mysterious deadlocks and other difficult-
to-debug problems, suggesting that shared-memory based designs may not be the best choice for cache-incoherent architectures such as the IMAPCAR2.

Pilot [10] is a simple wrapper around MPI [38], the Message Passing Interface for cluster computing. While a port of MPI (and thus Pilot itself) to the IMAPCAR2 is impractical and unnecessary, porting the Pilot API to this architecture would be worthwhile. By providing a simple, proven API — one that works regardless of the data types and alignment involved, automatically chooses the best way to manage mode switching and message passing, and provides high-level collective operations such as broadcasting that can automatically make use of hardware features where appropriate — the IMAPCAR2’s MIMD mode would become more accessible. Programmers could start programming faster, and more easily write programs that work the first time — and are easier to debug if they don’t — at the cost of some performance overhead.

1.2 Objectives

The AutoPilot parallel programming library ports the Pilot cluster computing API to the IMAPCAR2. By implementing a Pilot-like API on the IMAPCAR2, AutoPilot can not only make programming the IMAPCAR2 specifically an easier task, but also demonstrate the viability of Pilot’s programming model for multicore, cache-incoherent architectures in general. While AutoPilot itself is, of necessity, platform specific, the API and concepts behind it are applicable to more than just the IMAPCAR2 and its descendants.

The development of a complete, production-ready implementation is beyond the scope of this thesis; instead, a prototype is developed, demonstrating the implementation of core
Pilot features on the IMAPCAR2 and their use in programming. While there is great scope for improvement, particularly in performance, before it can be considered production-ready, this demonstrates the basic functionality of the AutoPilot library, and the viability of a Pilot-based programming model.

1.2.1 Thesis Statement

The implementation of Pilot’s simple, printf-inspired API and processes-and-channels abstraction will result in a simpler and safer API for multicore programming on the IMAPCAR2 — simpler in the reduced number of functions and types needed to accomplish parallel programming tasks, and safer in the presence of builtin safety checking mechanisms — without significantly compromising program size or performance.

1.3 Thesis Outline

Chapter 2 gives background information on the IMAP architecture and the Pilot library. This includes both the IMAP image processing design presented circa 199X, the history of the IMAPCAR chip family, and the specifics of the IMAPCAR2 which AutoPilot targets. It also discusses One-Dimensional C (“1DC”), the C variant used for programming on the IMAPCAR. Related hardware architectures and software are discussed in Section 2.5.

Chapter 3 and Chapter 4 discuss the design and implementation, respectively, of AutoPilot. The former covers design requirements, differences in the API compared to the MPI-based version, and the rationale for those differences, as well as an in-depth look at
the existing IMAPCAR2 parallel programming libraries, lib1dc and pthreads. The latter covers the actual implementation, including problems that arose during implementation and resulted in revisions to the design.

Chapter 5 analyzes the results. Microbenchmarks are used to assess the performance impact of AutoPilot functions relative to the IMAPCAR2’s standard message passing library; a case study, N-Queens, is used for a higher-level code size and performance comparison.

Finally, Chapter 6 summarizes the results of the research, and discusses possible areas of future work.
Chapter 2

Background and Related Work

The AutoPilot library makes the Pilot API available on the IMAPCAR2 processor and related, XC Core-based processors. It is not a true port of Pilot — there is no shared code — but rather, a port of the API. While originally designed for MPI-based supercomputing clusters, the Pilot API also maps well to the capabilities of the IMAPCAR2: it assumes no shared memory (the use of which is discouraged on the IMAPCAR2 due to cache incoherence), and all interprocess communication is performed via message passing (which has hardware support on the IMAPCAR2 via the ring network interconnecting the processors). Some collective operations, such as broadcasts, also have hardware support and equivalents in the IMAPCAR2’s system library.

This chapter gives a detailed background on the Pilot library itself, the IMAP architecture in general, the IMAPCAR2 processor family in specific, and the development tools — including 1DC, the custom C variant used for programming it — available for the IMAPCAR2. It also contains an overview of related hardware and software.
2.1 Pilot

Pilot [9, 11] is a cluster computing library created at the University of Guelph by J. Carter and Dr. W. Gardner et al. It is a wrapper around the Message Passing Interface [38] (MPI) commonly used in cluster computing, with a much smaller and simpler API and a number of additional features such as automatic deadlock detection. It is “a friendly face for MPI”, which can be deployed on any MPI-based platform to provide a set of communication functions with an API strongly inspired by C’s printf and scanf.

2.1.1 Basic Concepts

Pilot is conceptually based on C.A.R. Hoare’s Communicating Sequential Processes formalism [22] (CSP). Primarily, it is structured around processes and channels. A process is a single independent thread of execution; a channel, a line of communication between processes. These are the only means by which interprocess communication is permitted, a restriction that aids in program analysis and permits the use of debugging tools such as Pilot’s deadlock detector.

Channels form the sole means of communication between processes. Each channel is a unidirectional, one-to-one link between two processes; the direction of the link and which processes it connects are specified when the channel is created, and cannot be changed afterwards. In particular, this means that two-way communication between processes requires the creation of two channels, one in each direction. Channels are also partially synchronous: attempts to read from a channel will block until a message has been received, and attempts to write to one will block until all of the message data has been successfully
copied from the arguments passed to \texttt{PI\_Write}. Finally, channels are untyped; the type of each message sent is specified when read or write is called, and thus, one channel can be used to send any number of differently-typed messages in its lifetime. The API for sending messages via channels is \texttt{printf}-inspired, with a format string specifying the type of the message, and subsequent arguments the values to be sent.

Pilot also supports \textit{collective operations}, read or write operations involving multiple channels. These are performed on \textit{bundles}, collections of channels with a common endpoint; much like channels, the properties of each bundle — including which collective operation it is intended for — are specified when it is created and are fixed thereafter. \textit{Broadcast} allows the same message to be efficiently written to every channel in a bundle at once, while \textit{Scatter} and \textit{Gather} are used to efficiently write to (or read from) each channel a different message — provided all messages have the same type. Newer versions of Pilot also support \textit{Reduce}, which collects messages in a manner similar to Gather, but also reduces them to a single final value using a specified function. Finally, \textit{Select} allows the program to find an individual channel in a bundle which is ready for reading, so that a message can be read from it individually.

\subsection*{2.1.2 Program Structure}

Pilot programs, like MPI programs, are conventionally Single Program, Multiple Data — the same program runs on each node, and if different code needs to be run on each, it is the responsibility of the program to handle that. Unlike MPI, however, the actual details of choosing which function runs on which node in this usage pattern are handled automatically, and Pilot presents a more “\textit{pthreads-like}” interface in which the user creates
processes and assigns each a function to execute.

Like MPI, it is possible to use Pilot in a Multiple Program, Multiple Data configuration, in which a different executable is assigned to each node. This is rarely useful on homogenous clusters — it is easier to simply assign a different function to each process using the Pilot API — but is necessary on heterogenous clusters where multiple builds of the program are required for the different processor architectures involved.

A Pilot program is divided into two phases, the configuration phase and the execution phase. All setup — process, channel, and bundle creation — takes place during the configuration phase. Process creation consists of calling PI_CreateProcess and passing it a user function to execute and arguments to pass to that function. Channels and bundles are created in a similar manner, by calling PI_CreateChannel or PI_CreateBundle. During this phase, the user functions are not yet executing, and channels and bundles cannot be used. The configuration phase runs in parallel on all nodes.

Once all nodes have called PI_StartAll, and thus completed the configuration phase, the execution phase begins. At this point, one node becomes the main process (PI_MAIN), and returns from PI_StartAll to continue executing the user code that called it; the others are each assigned one process to execute and call the corresponding user function. (If more nodes are available than Pilot processes created, the excess nodes simply exit.) Once the execution phase begins, no new processes or channels are created: the overall structure of the program is fixed during the configuration phase, and there is no way to send messages to arbitrary processes in an ad hoc manner.

All of the actual work, including all interprocess communication via channels and bundles, happens during the execution phase. When all of the processes have finished
PI_MAIN has signaled completion by calling PI_StopMain, Pilot shuts down in good order, and leaves the main function to perform any further cleanup required before exiting completely.

2.1.3 Communication

All point-to-point communication in a Pilot program is done via channels, using the functions PI_Read and PI_Write. These use APIs modeled on fprintf and fscanf, taking a PI_CHANNEL as their first argument, and a format string, specifying the number and types of additional arguments, as their second. Subsequent arguments are the values to write (for PI_Write) or pointers to the variables that are to hold the values being read (for PI_Read). The supported format strings, and their meanings, are listed in Table 2.1. printf’s “field width” syntax is adapted here for arrays; a format string of "%u" means that the corresponding argument is (a pointer to) a single unsigned int, while "%16u" means that it is a pointer to an array of sixteen unsigned ints.

One-to-many and many-to-one communication follows the same pattern, using bundles instead of channels. PI_Broadcast is, apart from that difference, identical to PI_Write in invocation. PI_Scatter and PI_Gather expect arrays of values or pointers, respectively, but are otherwise the same. The biggest difference is probably PI_Reduce, which requires a different format string structure specifying not just the number and types of arguments, but also what reduction function to use for each one — with a number of predefined operations for common arithmetic and bitwise operations, and the option to provide a user-defined function.
<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d, %i</td>
<td>int</td>
</tr>
<tr>
<td>%ld, %li</td>
<td>long int</td>
</tr>
<tr>
<td>%lld, %lli</td>
<td>long long int</td>
</tr>
<tr>
<td>%u, %lu, %llu</td>
<td>unsigned (long, long long) int</td>
</tr>
<tr>
<td>%hd, %hi</td>
<td>short int</td>
</tr>
<tr>
<td>%hu</td>
<td>short unsigned int</td>
</tr>
<tr>
<td>%hhu</td>
<td>short short int</td>
</tr>
<tr>
<td>%b</td>
<td>raw byte (no number/character conversion)</td>
</tr>
<tr>
<td>%c</td>
<td>char, subject to character set conversion</td>
</tr>
<tr>
<td>%f</td>
<td>float (single precision)</td>
</tr>
<tr>
<td>%lf</td>
<td>double</td>
</tr>
<tr>
<td>%llf</td>
<td>long double (&quot;quad&quot;)</td>
</tr>
<tr>
<td>%m</td>
<td>MPI_Datatype, used for user defined types</td>
</tr>
</tbody>
</table>

Table 2.1: Pilot format string codes
2.1.4 Error Detection

Pilot performs some basic sanity checking of function preconditions at runtime. This includes checking that processes are calling read and write on channels that they are permitted to; checking that collective operations are being used on bundles with the correct usage; checking that the message the writer is sending is of the same type as the one the reader is expecting to receive; and, insofar as this is possible in C, checking that the number of values being sent in a message matches the number specified by the format string.

Pilot also includes an optional deadlock detector. When enabled, this creates a separate process which monitors all message-passing activity of the other processes. In this manner, it can detect deadlocks — such as when two processes are each waiting for a message from the other — as soon as they occur. While the performance overhead of the deadlock detector is considerable, it can report not only when a deadlock has occurred, but which processes are deadlocked, what channels they are blocked on, and even where in user code the calls were made that deadlocked them, making it invaluable for debugging.

Errors can also be reported by user code, by calling PI_abort. Like failed sanity checks and deadlocks, this causes all nodes of the program to immediately exit, outputting an informative error message.

2.1.5 Example Program

Figure 2.1 shows a simple Pilot program that makes use of both channels and bundles. Figure 2.2 shows the process/channel graph for this program. This program creates a number of worker processes (all running the same user function), broadcasts an int to all of
```c
#include <pilot.h>
#include <stdio.h>

#define NROFWorkers 4

PI_CHANNEL * toWorkers[NROFWorkers];
PI_CHANNEL * fromWorkers[NROFWorkers];

/* read an int from PI_MAIN, return that multiplied by our index */
int worker(int index, void * udata) {
    int x;
    PI_Read(toWorkers[index], "%d", &x);
    PI_Write(fromWorkers[index], "%d", x * index);
    return 0;
}

int main(int argc, const char ** argv) {
    /* Configuration phase */
    PI_BUNDLE * toWorkersBundle;
    PI_Configure(&argc, &argv);
    for (int i = 0; i < NROFWorkers; ++i) {
        PI_PROCESS * proc = PI_CreateProcess(worker, i, NULL);
        toWorkers[i] = PI_CreateChannel(PI_MAIN, proc);
        fromWorkers[i] = PI_CreateChannel(proc, PI_MAIN);
    }
    toWorkersBundle = PI_CreateBundle(PI_BROADCAST, toWorkers, NROFWorkers);
    /* run phase */
    PI_StartAll();
    PI_Broadcast(toWorkersBundle, "%d", 925);
    for (int i = 0; i < NROFWorkers; ++i) {
        int x;
        PI_Read(fromWorkers[i], "%d", &x);
        printf("Worker%d replied with value%d\n", i, x);
    }
    /* Shut down pilot */
    PI_StopMain(0);
    return 0;
}
```

Figure 2.1: A simple Pilot program
them, and receives from each in turn the result of multiplying that int by a number unique to each process.

Lines 1-4 are preamble, including necessary headers and defining the number of workers. Lines 6 and 7 declare global storage for the channels connecting PI_MAIN to the worker processes in both directions, in scope to both the workers themselves and main. Note that no global storage is allocated for the PI_PROCESS pointers themselves; there is no reason for the worker function to address those, as the only operations it could usefully perform on them are restricted to the configuration phase.

The worker function spans lines 10 through 16. The function signature on line 10 is typedef’d as PI_WORK_FUNC in pilot.h and is the signature required for all Pilot process user functions; index and udata are arbitrary values specified in the configuration phase when each process is created. Line 12 reads the number broadcast from PI_MAIN —
PI_Read is used for all individual-channel reads, regardless of whether the other end is on its own or part of a bundle. The following line sends the result back to PI_MAIN. Note the use of a scanf-like interface for the read, and printf-like for the write. Finally, line 15 returns a status code to Pilot (which is logged, but has no significance to Pilot itself).

main occupies the rest of the listing. Lines 19–29 are the configuration phase. It starts with the call to PI_Configure, passing in pointers to the argument array so that Pilot can scan for and remove Pilot and MPI specific arguments. Lines 23–27 then create the processes and channels — each process (line 24) is passed a unique index, which corresponds to the index in the toWorkers and fromWorkers arrays of the channels connected to that process. The channels to (line 25) and from (line 26) the process are then created and stored in that same index in the appropriate arrays (the actual PI_PROCESS * is stored in a temporary variable, as it is unneeded after this point). Finally, on line 29, the bundle is created and configured for broadcasting.

On line 32, the configuration phase ends and the execution phase begins with the call to PI_StartAll. At this point, the workers start up and immediately block on PI_Read. The main process (also known as PI_MAIN) then calls PI_Broadcast on line 34 to transmit the test value (an arbitrarily-chosen 925) to the workers. It will be picked up in each worker by the PI_Read on line 12.

Finally, PI_MAIN iterates over all of the channels leading from the workers to the main process and collects the result from each one using PI_Read, on line 38. (The collective operation PI_Gather could also be used here, to collect the results from all processes at once). Having done this, it calls PI_StopMain, passing it an exit code (which is logged) and blocking until all of the workers have returned, at which point it returns and, immediately
afterwards, the program exits.

The above “classic Pilot” API was able to be recreated more or less intact on the IMAPCAR2. For design reasons, some differences exist, as will be explained in Chapter 3.

### 2.2 The IMAP Architecture

IMAP, the Integrated Memory Array Processor, is a processor architecture developed by Fujita et al. [17] in 1994 for high-performance image processing applications. Intended to be used as a coprocessor to a normal central processing unit, it consists of an array of 8-bit Reduced Instruction Set Computer (RISC) SIMD Processing Elements (PEs), connected to each other with a ring network for fast communication between adjacent PEs, and to the controlling CPU via a tree network for dispatch of instructions. In addition, each PE is directly connected to 2KB of PE-specific on-chip memory (image memory or IMEM), used to store the data that the PEs operate on. Early IMAP designs ranged from 8 to 128 PEs per chip, with 1KB to 32KB of IMEM per PE [18].

In use, each image to be analyzed is loaded into the IMEM, typically in such a way that each row or column is held completely within a single PE’s memory. In this way, each PE can operate on its own slice of the image without ever needing to access main memory. In cases where information about adjacent pixels is needed, the ring network connecting the PEs (the \textit{N-ring}) can be used to exchange information with the PEs handling the adjacent rows or columns.

This design addresses the main bottleneck with other SIMD processors of the era, which was access by the SIMD PEs to external memory — requiring a large number of
Figure 2.3: IMAP-VISION board configuration [16]

connections to permit parallel access to off-chip RAM. It also makes it easy to expand a system’s image processing capabilities by connecting multiple IMAP processors in series.

2.2.1 IMAP-VISION

The successor to the original IMAP prototype, IMAP-VISION, was a commercially available platform for PC-based video processing. It consisted of a PCI expansion board containing eight IMAP chips — each one containing 32 8-bit PEs, 32KB total IMEM, and attached to 16MB of off-chip SDRAM — a set of digital-to-analog and analog-to-digital converters for video output and input, respectively, and a central RISC processor to manage the IMAP array and execute sequential code. The external memory was added to address the greatest weakness of the original IMAP prototypes, which was a lack of available memory for the PEs to use for space-intensive algorithms [16].
The IMAP-VISION also came with a complete development environment, including a debugger, simulator, profiler, and code editor. It used One-Dimensional C (1DC), a C variant with extensions for making effective use of the IMAP processor array and its IMEM. 1DC is described in more detail later in this thesis, in Section 2.3.

Tests using the IMAP-VISION to implement a 10 GOPS (giga-operations per second) highway vehicle detection and tracking system showed it capable of operating at 15 frames per second (66ms/frame or less), running an algorithm with a 95% or better detection rate and a false positive rate of 6% or less [32].

2.2.2 IMAP-CE

The IMAP-CE, Integrated Memory Array Processor for Car Electronics, was the IMAP-VISION’s descendant. Released in 2003, it greatly compressed the design: a single chip now contained 128 PEs, 2KB of IMEM per PE, and the 16-bit controlling processor itself (now called the CP). It also increased the clock speed to 100MHz and improved the execution pipelines of the PEs and CP, allowing it to reach 51.2 GOPS at full clock speed [31, 34].

Like the chips making up the IMAP-VISION, individual IMAP-CE chips could be interconnected to expand the width of the SIMD processing array and the amount of IMEM available. The development environment was largely unchanged from the IMAP-VISION.

Benchmarks using the vehicle detection system described in [32], and later benchmarks using other algorithms [31], showed that the IMAP-CE outperformed both the IMAP-VISION and contemporary general-purpose processors. Furthermore, while slower than other dedicated media processors of the time (based on fast fourier transform benchmarks),
it was also smaller, cheaper to manufacture, and required less power and a slower clock rate. The developers identified the 8-bit memory access width of the PEs as a likely bottleneck (the other three processors compared were all 16-bit).

At this point, the single-chip design of the IMAP-CE is a recognizeably close relative of the IMAPCAR2.

### 2.2.3 IMAPCAR

The IMAPCAR was an overhaul of the IMAP-CE design, intended for commercial vehicular vision applications and released in 2008. The fundamental design concepts were unchanged, but a large number of small improvements were made [33]. Some of these changes — temperature tolerance, error correction code support for program and data memory, and parity checks on image memory — were made to satisfy the safety requirements for use in vehicles. Others were improvements in cost, such as reductions in process size, transistor count, and power usage. It also saw improvements in Direct Memory Access (DMA) hardware, hardware support for image buffer scaling and cropping during DMA operations, and dramatic improvements to the video pipeline, allowing it to process up to three 512 pixel wide or two 768px wide video streams (compared to the IMAP-CE’s 2x256px or 1x512px limit).

Overall, the IMAPCAR is reported to be capable of 100 GOPS with no clock speed increase relative to the IMAP-CE. Performance evaluations [44] support this. The developers noted, however, that the IMAPCAR remained ineffective at exploiting task-level parallelism, and that doing so is an important next step — one that cannot be accomplished by simply adding more PEs (which only help with uniform data-level parallelism).
possibility of adding secondary CPs was considered and rejected, as the increased transistor count and die size would seriously impact the IMAPCAR’s power, space, and cost efficiencies.

The IMAPCAR and IMAPCAR2’s intended usage as processors for automotive electronics was the genesis of the name “AutoPilot”; it is a combination of automotive (as these chips are most widely used in the automotive industry) and autonomous (as they can and have been used to implement limited self-driving capabilities) with the name of the Pilot library from which it takes its API and overall design.

2.2.4 IMAPCAR2 and the XC Core

The IMAPCAR2 is a processor developed by S. Kyo et al. to address the IMAPCAR’s weakness at exploiting task-level parallelism. The microarchitecture it is based on is called the eXtensible Computing Core or XC Core, and is a heavily modified and upgraded IMAPCAR [30]. Specifically, it adds dynamic reconfiguration capability that allows software to select, at runtime, a tradeoff between SIMD and MIMD operation.

The obvious way to address MIMD support — as discussed in previous papers by the designers [33] — is to simply add more CPs to the chip. However, this would dramatically increase chip size, transistor count, and power requirements. Further inefficiencies arise from the fact that in SIMD-heavy operation, at least some of the CPs would not be used to their full potential, whereas in MIMD-heavy operation, the SIMD PEs would be idle.

The approach taken instead was to add control hardware that would permit each PE to act either as a SIMD processing element, as in earlier IMAP processors, or as a component of an additional, CP-equivalent processor core. This is accomplished by arranging the
PEs into groups of eight called tiles, each one of which consists of an upper and lower half of four PEs which are interconnected with initially dormant reconfiguration hardware. When activated, each of these four-PE groups begins to act in concert as an additional CP, called a processing unit or PU — the directly-attached IMEM becomes data and instruction cache memory; the registers become cache tags; and the excess arithmetic/logic units and registers become the single-precision floating-point unit [29]. Figure 2.4 shows a diagram of how a four-PE tile can be reconfigured as a PU. Note that D$ and P$ stand for data cache and program instruction cache, respectively. The PUs support a large subset of the CP’s instruction set, and code compiled for the PUs can also execute on the CP. To support this, the PEs themselves have also seen an upgrade to a 16-bit data path. Furthermore, the PUs are interconnected with two ring
networks, the *C-ring* and the *M-ring* — the former allowing them to pass 16-byte messages directly between PUs (including the CP), the latter connecting the PUs to the DMA hardware used for external memory access. Tiles can be activated either in their entirety, resulting in a multicore system with no SIMD hardware, or only the bottom half of each tile can be activated, leaving half of the SIMD array active and usable by the CP.

On the software side, surprisingly little changed, with 1DC still the language used for programming the IMAPCAR2. There are new library functions used for communication between PUs, and for switching between SIMD and MIMD modes (the latter called *MP mode*). The most dramatic change is the introduction of an Eclipse-based development environment with an integrated simulator and debugger, superceding the old TCL-based development environment.

**Memory Model**

The memory model of the IMAPCAR2 is somewhat unusual. It can essentially be divided into four types: *program memory* (PMEM), *data memory* (DMEM), *external memory* (EMEM), and *internal* or *image memory* (IMEM).

In SIMD mode, this is relatively straightforward. PMEM is the read-only program memory that contains executable code, and occupies the bottom of the memory map. DMEM contains the stack and heap for the CP (and occupies whatever portion of the lower 16MB of memory is not dedicated to PMEM). EMEM is the rest of the externally-connected SDRAM. And finally, IMEM is the array of 2K buffers directly attached to the PEs. The SIMD memory map is shown in Figure 2.5. Here, $I$ is used for the program instruction cache.
In MP mode, it gets slightly more complicated. IMEM is gone (that memory is now being used for the data and instruction caches on the PUs), but there are multiple DMEM areas: each PU is assigned a memory region when it starts up, which it uses for stack and heap. In typical usage, this is allocated from the CP’s DMEM (either as a static declaration, or using dynamic allocation). Figure 2.6 shows details of the PU DMEM layout, and the location of the DMEM regions in CP memory.

Furthermore, while each PU can only address 16MB of memory at a time, it does not always have to be the same 16MB. Each PU has a register that controls which 16MB “page” of RAM it addresses; by default, all PUs address the same “page 0” that the CP does. This can be changed, but doing so raises a host of issues, which are discussed in Section 3.1.
This is further complicated by the fact that there is no cache coherency hardware — which is to say, there is no inbuilt support for keeping the caches attached to different processors in sync. For example, if two processors have the contents of the same variable in cache, and one updates it, the other will not see that update, but will continue to use the old value it cached originally; if both update it, they will each see their own update, and it is unspecified which update “wins” when the contents of the caches are copied back to main memory — which will not happen automatically until cache pressure requires the replacement of that cache line.

While the caches can be bypassed entirely, this is not recommended, as cache access is many times faster than access to external memory, and is not subject to bottlenecks, as when multiple processors are accessing the same external memory. The lack of cache coherency, however, means that extreme caution is required when using shared memory,
and it is preferred to avoid it entirely and use message passing instead.

In mixed mode, both models are in effect. IMEM is still available for those PEs that are still active; the active PUs each have a DMEM segment in their assigned memory page.

**Performance Analysis**

A performance analysis of the IMAPCAR2 [42] discusses typical usage, and why MIMD support is necessary. The software being used for the evaluation — a white line detection algorithm — consists of seven processing stages. Of these, the three responsible for 99% of the processing time were chosen for benchmarking: extraction of potential white line areas, connection of these areas into white line segments, and verification of segments. Of these, the first two are amenable to SIMD execution, and the last is not. This structure — where SIMD execution can be used to identify *Regions Of Interest* (or ROIs), which must then be analyzed more thoroughly in ways that do not lend themselves to SIMD operation — is typical of image-processing applications for which the IMAPCAR2 is used.

The results of the analysis confirm the value of the IMAPCAR2’s mode switching capability. A pure SIMD approach, in which region of interest analysis is performed sequentially by the CP, takes 39ms/frame. The introduction of mode switching and the use of a task-parallel design for the third stage of the algorithm, however, reduces the cost of that stage by a factor of nearly 12, and the performance as a whole to just over 25ms/frame, well within the 33ms/frame limit for realtime execution. These results are reprinted in Figure 2.7.
2.3 1-Dimensional C

IMAPCAR and IMAPCAR2 programming uses 1-Dimensional C (1DC) [2], a variant of the C programming language with extensions specifically for effective use of the IMAPCAR2’s SIMD capabilities. Primarily, it permits the declaration of arrays as `sep`, indicating that their storage should actually be placed in IMEM; loads and stores to array contents will be directed to IMEM rather than main memory, and there are ways of specifying parallel operations on these arrays that will cause them to compile to SIMD code that makes full use of the PEs.

The 1DC extensions are concerned primarily with SIMD operations; most of the extensions do not apply when in MP mode, and are either ignored or cause a compile error or
runtime exception, depending on context. As AutoPilot is concerned solely with MP mode, most of the 1DC extensions are irrelevant to it.

### 2.3.1 New Declarations

1DC adds a number of new modifiers to C variable declarations — uni and multi, outside, align, and common and separate. Of these, the only ones relevant to MP mode programming are outside and align.

uni and multi are used to declare where the data is allocated: in EMEM or IMEM. uni, the default, behaves like normal C and allocates storage in EMEM; multi allocates storage in IMEM, and emits code to initialize it correctly on all of the PEs.

common and separate are used to indicate whether the variable has the same value on every PE, or whether each PE’s value for it can vary independently of the others — in effect, whether to allocate a single shared variable, or to duplicate it across the entire PE array. Declaring a variable separate implicitly declares it multi as well, as it doesn’t make sense to have a variable that can have per-PE values which isn’t actually allocated in PE IMEM.

align is used when declaring global arrays, and forces them to be aligned to a cache line boundary (which, in practice, means aligned to an address that is a multiple of 512). In MP mode, this is helpful for some functions that require specifically aligned memory regions; a few, however, have even stricter alignment requirements (such as 1024-aligned) which align is not guaranteed to satisfy.

Finally, outside forces the variable to be allocated in non-cached EMEM, meaning that all access to it bypasses IMEM or the cache entirely. While this results in a noticeable
performance penalty, it also means that it is always coherent across processors (barring race conditions). It is worth noting that outside is a property of the symbol, not the storage; in particular, casts can be used to add (or remove) the outside qualifier of a pointer, and whether accesses to the underlying storage bypass the cache or not depends on which pointer one accesses it through.

The IMAPCAR2 SDK also automatically provides a number of typedefs for numeric types of guaranteed width. These follow the same convention as the standard POSIX `<stdint.h>` header: the word “int”, prefixed with “u” for signed types and suffixed with the width, in bits of the type — so, for example, int16 is a 16-bit signed int, and uint32, a 32-bit unsigned one.

In addition to the basic signed and unsigned types, there are typedefs for separate, outside, and outside separate types. The former is denoted by a “sep_” prefix; the latter, by “o” - so sep_int32 is the type of a separate 32-bit signed integer, ouint8 is an outside 8-bit unsigned int, and osep_int16 is an outside, separate signed 16-bit int. Like the POSIX definitions, these provide a reliable (and portable) way of declaring variables of specific width, regardless of how standard C modifiers such as short, long, and long long affect the data width of numeric types.

### 2.3.2 New Operators

1DC also adds a set of new operators for SIMD operation, used for inter-PE communication via the N-ring and selection, modification, and logical reduction of sep values. All of these operators are based on related C operators prefixed with :.

The :< operator is used for N-ring communication. :< x evaluates to the value of the
sep variable \( x \), as seen by the PE to the immediate right (i.e., the one with the next highest PU ID, modulo the number of PUs). \( x :< n \) is used for accessing PEs further away; it is the value of \( x \) on the PE \( n \) PEs to the right. \( :> \), correspondingly, accesses the PEs to the left. The \( :[ \) and \( :] \) operators — which are always used in a pair, bracketing a subscript — are another means of accessing data on other PEs; \( x :[n:] \) is the value of \( x \) on the PE with (absolute) ID \( n \).

\( :\| \) and \( :\&\& \) are the logical reduction operators. Unlike \( || \) and \&\&, they are unary, each operating on a single sep expression. The former is true if the value of that variable is true on any PE; the latter, if it is true on all of them. The operator argument is not limited to simple variables; it can be more complex expressions dependent on sep variables. For example, \( :\&\&(x > 5) \) is true if and only if \( x > 5 \) on every PE.

\( :([ :) \) — which, like \( :[ :]) \), is always paired — is used for initialization. \( x = :((1, 2, 3 :) \) assigns 1,2,3 to the first three PEs and 0 to the rest. \( x = :((1, 2, 3, :) \) assigns 1 and 2 to the first two and 3 to the rest. \( x = :((1:4, 5, :) \) assigns 1 to the first four and 5 to the rest.

Finally, \( := \) is used for assignment. \( x = y \) copies \( y \) to \( x \) on the CP. \( x := y \) copies \( y \) to \( x \) on every PE, using the PE-local values for \( x \) and \( y \).

### 2.3.3 New Keywords

The keywords added by IDC are all related to context selection, i.e., choosing which PEs will execute a given block of code. (As it is not possible for different PEs to execute different instructions simultaneously, PEs that do not pass the context check are disabled for the duration of the block.) They are the standard C flow control keywords — if, else,
while, do, and for — with "m" prefixed, for example, mif. The expression the flow-control keyword depends on must always be sep (as otherwise it would have the same result for all PEs). mif \( (x > 5) \{ \text{foo}(); \} \text{ else } \{ \text{bar}(); \} \), for example, would call \text{foo} on all PUs for which \( x > 5 \) (while the others idle), and then call \text{bar} on the rest. Like the new operators, these keywords are only meaningful in SIMD mode.

Keywords also nest — an mif block nested inside another mif block, for example, will execute only on those PUs that pass both tests. It will never activate a PU that was deactivated by the outer test. There is one exception, mifa and melsea (the "a" standing for "all"); these ignore nesting entirely, but are faster than the versions that respect nesting.

Internally, these keywords are implemented similar to branching keywords on modern programmable GPU platforms (such as CUDA). While it is not possible for different PEs to execute different instructions concurrently, it is possible for them to be selectively disabled; thus, when a branching instruction such as mif is used, all of the PEs that do not meet the condition idle while the ones that do execute. This allows greater flexibility in writing SIMD code, at the cost of increased cycle count when executing branches.

## 2.4 The IMAPCAR2 SDK

The IMAPCAR2 Software Development Kit consists of a set of tools and libraries for developing IMAPCAR2 applications. The core of the SDK consists of a runtime library, \text{lib1dc}, and a set of command-line, Microsoft Windows-hosted development tools. The library serves the same role as libc on other platforms, providing the basic set of functions and data structures needed to write programs targeting the chip (including a large number
of functions not present in the ANSI C standard, providing access to chip-specific features). The development tools include a cross-compiler for 1DC source, an assembler and linker, a library archiver, and a simulator, debugger, and profiler. The IMAPCAR2 toolchain uses custom object code and executable formats, the common ar format for static libraries, and does not support dynamic linking.

The simulator and profiler are used for offline debugging, testing, and analysis of IMAPCAR2 programs, and provide a simple command-line interface; they can simulate a variety of chip configurations. Programs running in the simulator can load video “input” from video files or static pixmaps on disk. The debugger is used to control a physical test board, allowing new programs to be loaded onto it and debugged as they run. Both of these permit a program to be loaded and run at “full speed” or stepped through instruction-by-instruction, as well as other standard debugging features such as program disassembly, memory and register access, and break-, watch-, and tracepoints.

A graphical development environment is also provided, built on top of these tools and based on Eclipse [15]. It features a 1DC project class that automatically invokes the compiler, linker, and archiver as needed, and IMAPCAR2 debugging modes that provide a graphical interface to the simulator and debugger. It is also capable of using the debugger to provide source-level debugging of IMAPCAR2 programs.

Finally, a number of other utility libraries, including utilities for video and image processing and a pthreads implementation, are included, along with source code for these libraries and for lib1dc itself. This allows programming the chip in either a “low-level” mode that directly exposes the basic capabilities of the hardware, or at a higher level using provided convenience functions for common image and video analysis tasks and thread
management. The pthreads library is discussed in more detail in Section 3.2.

2.4.1 lib1dc

The 1DC Standard Library, lib1dc, serves as both the C standard library and system library for IMAPCAR2 programs. In addition to most of the ANSI standard C functions (provided in the usual headers like <string.h>), it provides a large set of functions for accessing the specialized features of the chip in both SIMD and MP modes. The manual [3] classifies these into memory functions, for allocating and freeing memory and transferring data and regions of interest between different memory spaces (EMEM, IMEM, and DMEM); operation functions, for fast SIMD bit manipulation, logic, and math; and MP mode functions for managing PUs and sending messages between them. All functions in lib1dc that are IMAPCAR2-specific (i.e., not part of the C standard library) have their name prefixed with IX, IX_, or IX_; those starting with (uppercase) IX or IX_ can only be called safely from the CP, while those starting with (lowercase) ix_ can be called either from the CP or (in MP mode) the PUs.

The memory functions consist primarily of many variations on malloc, free, and memcpy, for allocating and freeing in and copying between EMEM, IMEM, and DMEM, and copying with or without respecting the cache. There are also provisions for performing simple transforms on the data as it is transferred to accommodate different pixel formats, and, in the case of the ix_roi functions, for copying a rectangular subsection of an image buffer into a smaller, linear buffer (or vice versa).

The operation functions consist of a large library of math, logic, and basic image manipulation functions; in essence, the basic set of data manipulation operations for SIMD
mode. These are all intended for operation on separator data from the CP in SIMD or mixed mode, and thus are not relevant to AutoPilot, which is concerned solely with MP mode.

Finally, the MP mode functions — which AutoPilot is built upon — fall into a few categories. There are CP-specific functions for controlling the PUs, both at a low level (\texttt{Ix\_fork*})\textsuperscript{1} and a high level (\texttt{Ix\_start*}). There are memory management functions, for manually managing cache writeback and invalidation (sometimes a necessity, given the lack of hardware-supported cache coherency) and for performing bulk memory copies, both with and without cache usage. Finally, there are message passing functions, for sending several different types of messages between the CP and PUs using the C-ring. These are discussed in more detail in Section 3.1, which looks at the process of writing MP mode functions on the IMAPCAR2 using the lib1dc API in more detail.

\section{Related Work}

\subsection{Hardware}

The IMAPCAR2 is far from the only architecture designed for automotive vision applications. While there is some commonality between them — they are all designed around the constraints of providing real-time video processing while minimizing power requirements, manufacturing costs, and size — there are a wide variety of design approaches. These can be roughly ordered based on the ratio of operational to control circuitry, with general purpose processors having a high proportion of the latter and application-specific integrated circuits (ASICs) consisting almost entirely of the former. In general, systems

\textsuperscript{1}The \texttt{*} is used as a wildcard here and elsewhere; that is, \texttt{Ix\_fork*} denotes the set of all functions with names beginning with \texttt{Ix\_fork}. 
with less control circuitry achieve greater efficiency, at the cost of flexibility.

The following is by no means an exhaustive look at all processors of this type, but it gives a representative cross-section of those chips that bear the closest relation to the IMAPCAR2. It is worth noting that while some processors, such as the Visconti [46], combine SIMD and MIMD capabilities on the same chip, the IMAPCAR2 appears to be unique in using the same hardware for both and switching between both modes at runtime, as opposed to providing separate hardware for each and making both available at all times.

The Xetal [26] and Xetal-II [6] are pure SIMD architectures, similar to the IMAPCAR. They are designed for high performance operation at low power levels; the Xetal-II has 320 SIMD processing elements and delivers 107 GOPs with a 600mW power supply. Like the IMAPCAR2, these chips feature on-chip image memory (called frame memory by the designers), directly attached to the PEs — 32Kb per PE, in the case of the Xetal-II. The programming model for the Xetal was assembly language with specialized SIMD instructions. By the Xetal-II, this was replaced with a C variant called eXTended C (XTC), which, like 1DC, consists of ANSI C with extensions for managing the SIMD capabilities of the chip, such as the vint type, which is analogous to 1DC’s sep int. The Xetal-II’s successor, the Xetal-Pro [43] [20], is designed for wide input voltage scaling and is capable of operating on as little as 0.4V; a number of internal changes support scaling down gracefully to low input voltages, but the programming model is unchanged from the Xetal-II. The developers have noted that the lack of support for task-parallel operation is a weakness of the architecture, and have mentioned the possibility of improving this via reconfiguration capability [7].

The MorphoSys processor [36] takes a similar approach, combining a central TinyRISC [8]
processor with an 8x8 grid of programmable Reconfigurable Cells or RCs. Each row or column of the RC grid can be configured independently of the others, and configurations can be loaded into the “context memory” which holds RC configuration data concurrently with RC operation, upon which the RCs will switch operating mode on the next clock cycle. However, the RCs are fundamentally limited to SIMD operation; reconfiguration consists of choosing an operation to execute, and locations (memory, or other RCs) of operands and results. This permits some degree of pipelining (by configuring each row or column with a different operation, and chaining them together), but not general-purpose MIMD operation. It has been used to implement the TwoFish block cipher [37].

Visconti [46] takes a non-reconfigurable hybrid approach, integrating three RISC processor cores (Media Embedded Processors or MePs) on a single chip. Each core operates independently of the others, but includes a 3-way 64-bit SIMD coprocessor for high-performance data-parallel operation, supporting instructions for convolution and other common image processing operations, in addition to general purpose SIMD instructions. The chip also contains a hardware affine transformation accelerator with an integrated image data cache, shared between the three MePs. This design results in an ability to exploit task-level parallelism by running a different thread on each MeP, but the performance in SIMD operations suffers compared to the IMAPCAR. Information on the Visconti’s programming model, unfortunately, is sparse.

FAVIS [41] combines a single Hitachi SH7055F RISC processor with a custom chip, CarVchip, which contains hardware support for binarization, filtering, and edge detection operations. This sits between the SH7055F and the video input, allowing fast, low-power preprocessing of the image. This allows performance comparable to the IMAPCAR on cer-
tain operations, with a smaller footprint and power requirements, at the cost of flexibility: the CarVchip is special-purpose hardware that cannot execute any algorithms except those designed into it, rather than being a fully programmable SIMD array.

### 2.5.2 Software

Except for AutoPilot, there is a dearth of other parallel programming software available for the IMAPCAR2. The two parallel programming libraries it does come with — lib1dc and pthreads — are discussed in Chapter 3. Information on programming models for similar hardware architectures is likewise sparse. On some platforms, it is unnecessary; FAVIS, for example, handles all parallel computation using specialized ASICs that the user does not (and cannot) program for.

T. Kodaka et al describe [27] a parallel programming environment for an architecture closely related to the Visconti (it uses the same MeP architecture, but more of them). This design relies on the data dependencies between threads being specified up-front, and the dependency graph being acyclic; given these constraints, it can avoid scheduling any thread until all of its dependencies have been satisfied, and once a thread has been scheduled, execute it on the next available core without ever suspending it. This not only avoids the overhead of task switching as threads block and unblock, but lets the library do away entirely with the mechanisms needed to support suspending an executing thread and safely resuming it later.

For parallel programming in general, of course, there is no shortage of theoretical models and practical libraries.

The Candidate Type Architecture [45] is probably the most widely used model for
analyzing parallel architectures; it models a parallel system as some number $P$ of sequential processors, each one with some amount of local memory accessible in unit time, and interconnected with the other processors via a network of unspecified topology. Access to non-local memory must be performed via this network and is assumed to be expensive, taking $\lambda$ time (where $\lambda > 1$). One processor, $P_0$, is considered to be a control processor responsible for mediating between the other processors (if necessary) and controlling access to off-chip resources. The CTA describes the IMAPCAR2 very accurately: the CP is the controlling processor, the PUs are the other processors, the caches are local memory, the C-ring and M-ring are the processor interconnect, and accessing EMEM is, indeed, quite expensive compared to cache access.

The PRAM model (which the CTA was introduced to address the flaws in) takes a simpler approach: all processors are connected to a shared memory, which they can simultaneously access in unit time. Concurrent reads of the same location are permitted; whether concurrent writes are, and if so, what happens, varies depending on the discussion. For the purposes of this discussion, parallel programming interfaces can be divided into PRAM-inspired shared memory or lock-modify-unlock systems, and CTA-inspired message passing or send-receive-reply systems. (There are other approaches, such as transactional memory [21], but they are not widely enough used in embedded systems to be relevant to this discussion.)

Shared-memory systems rely on a shared memory space which all processes can access concurrently. Interprocess communication is done by updating global structures that all of the communicating processes (for example, all of the threads in a single operating system process) have access to; synchronization — including the protection of such structures
from concurrent updates — is done using global primitives such as mutexes and condition variables. The POSIX Thread Library, pthreads [23], is a widely used parallel programming API of this type, widely supported on UNIX and UNIX-like operating systems, with ports existing for other OSes and platforms (including the IMAPCAR2 itself).

Message-passing systems, in contrast, do not make any assumption of shared memory, and instead provide a mechanism for threads to communicate by sending messages to each other. These mechanisms serve as both communication (transferring data in the payload of each message) and synchronization (controlling when threads run, or do not, by waiting for the receipt of messages). This is a common approach for distributed and cluster computing, in which processes are often running on physically distinct computers; it is used by MPI, as well as distributed computing languages such as Erlang.

There also exist hybrid systems, which rely primarily on message passing but also permit direct nonlocal memory access by processes. Thoth [12] is one such example: interprocess communication is handled via the primitives .Send, .Receive, .Reply, and .Forward, but processes belonging to the same process team share an address space, and may meaningfully pass pointers through messages. The lib1dc API that is the basis for IMAPCAR2 programming also uses this approach, providing functions for message passing between PUs, but also permitting nonlocal memory access either directly (by accessing the contents of variables stored outside the PU’s DMEM area) or indirectly (by using transfer functions, such as the ix_memcpy or ix_roi function families, which read or write from arbitrary memory addresses).

AutoPilot — unlike MPI or Pilot itself — is also a hybrid system. It places a greater emphasis on message passing than the lib1dc API, but still permits (and, indeed, cannot
prevent) nonlocal memory access. Internally, nonlocal access is used for some types of interprocess communication.

**CellPilot (and other Pilot variants)**

The most closely related project to AutoPilot is probably CellPilot [19], a Pilot extension developed by N. Girard to support the Cell Broadband Engine processor. While the Cell BE is most famous for its use in the PlayStation 3, CellPilot was primarily designed to support heterogenous supercomputing clusters which include both normal processors and Cells, such as the prickly cluster of SHARCNET, the Shared Hierarchical Academic Research Computing Network — clusters which have gone underused due to the difficulty in making effective use of Cell processors in Pilot or MPI based programs.

The Cell itself has an architecture reminiscent of the IMAPCAR2 in MP mode. Like the IMAPCAR2, it has a central processor (POWER-based, called the Power Processing Element or PPE), and a number of coprocessors called Synergistic Processing Elements which, like the PUs, are connected to the PPE via a high performance ring network. Each SPE is a RISC processor designed for SIMD operations, and has 256k of local memory directly attached to it. The similarities end there, however. Unlike the IMAPCAR2, there is no support for reconfiguration; the SPEs are always available, and their support for SIMD operations means they can used for either SIMD or MIMD tasks as desired. They are also significantly more powerful than the IMAPCAR2’s PUs, and the gap in capabilities between the PPE and SPEs — which must be programmed completely differently — is much greater than that between the CP and the PUs (which can, for most purposes, be treated interchangeably). The local memory on the SPEs is also not treated as cache, but as
a separate memory space which must be explicitly addressed, avoiding the illusion of cache coherency that the IMAPCAR2 can present to the unwary. Furthermore, while they both have DMA hardware, the Cell’s must be explicitly controlled by the programmer, while the IMAPCAR2’s is automatically used for off-chip memory access.

Unlike AutoPilot, CellPilot is based on the original Pilot code and is designed to interoperate with it; a CellPilot-based program can freely mix normal MPI nodes, Cell PPEs, and Cell SPEs, with processes on each communicating with any other. This requirement to be compatible with normal Pilot imposes constraints on CellPilot’s design which are not shared by AutoPilot; however, CellPilot also has more resources to work with, between the much greater amounts of memory available, the faster processors, and the lack of real-time video processing requirements.

CellPilot is not the only Pilot variant. Standard Pilot is written in C, but includes a compatibility layer for use by Fortran; in addition, there is an object-oriented C++ wrapper, called Pilot++, developed by P. Garrity and R. Brown at St. Olaf College. An earlier work by this author is LuaPilot [25], a wrapper for Pilot that makes is usable from the Lua scripting language. All of these are language bindings for Pilot: they provide a translation layer around the C library that implements Pilot, allowing it to be used from another programming language. Additionally, D. Recoski at the University of Guelph has developed Pylot [35], a Pilot workalike written in Python; unlike the bindings mentioned previously, this implements a Pilot-compatible API using the Python MPI libraries, rather than wrapping the original Pilot library. None of these, however, port Pilot to other architectures; they are all concerned with making Pilot accessible on the same platforms, but from other programming languages.
AutoPilot itself is, like Pylot, a workalike rather than a binding: it re-implements the same API using a different underlying implementation, rather than providing a translation layer around the original Pilot library.
Chapter 3

Design

AutoPilot’s goal is to make the Pilot API available on the IMAPCAR2. It acts as a replacement for the existing parallel programming APIs for the IMAPCAR2 — the low-level communication functions provided by lib1dc [3], and the POSIX threading API provided by libpthreads [5].

Of note is that unlike other Pilot projects — including CellPilot — there is no intent for AutoPilot to interface with other Pilot implementations, or off-chip processors. While CellPilot’s goal is to make the Cell convenient to use in high-performance clusters, AutoPilot is targeted at the same embedded-systems domain as the IMAPCAR2 itself. Consequently, there is no need for the design to be concerned with compatibility with MPI or other Pilot implementations.

The goal of AutoPilot, then, is to implement the Pilot API on the IMAPCAR2, making modifications to the API as needed to support the different capabilities and requirements of the IMAPCAR2. This chapter describes the design process to meet that goal; it starts with an overview of issues faced by programmers when using the lib1dc API (Section 3.1), followed by an examination of the IMAPCAR2’s pthreads implementation (Section 3.2). It concludes with an overview of the design for AutoPilot itself, and how it addresses those issues.
3.1 Programming Issues

There are three major issues with parallel programming on the IMAPCAR2 using the functions provided by the standard lib1dc library: process creation and management, interprocess communication, and global memory safety. This section discusses all three.

Memory paging is a fourth potential issue, but in practice one that rarely arises. Multiple pages are only needed when some processes require so much memory that it is no longer possible to fit the DMEM regions for all processes into a single page. Given that each page is 16MB, that DMEM does not need to be shared with the frame buffers, and that application developers are encouraged to keep each thread’s working set below 4K (to fit in cache), this is unlikely. AutoPilot does not presently support multi-page configurations, and this is discussed further in the section on future work.

3.1.1 Process Creation and Management

lib1dc provides two primary mechanisms of different levels of abstraction for creating processes. Both of them, however, fundamentally consist of activating one or more PUs, providing them with a DMEM region and process entry point, and then starting them.

The low-level interface for PU activation and process creation provided by the library consists of the \texttt{I\_start\_mode} and \texttt{I\_fork*} functions. \texttt{I\_start\_mode} is used to switch the processor between SIMD, mixed, and multicore modes; once the system is in mixed or multicore mode, \texttt{I\_fork\_init} is used to configure each PU, \texttt{I\_fork\_d} to prefill the data cache, and \texttt{I\_fork} to specify an entry point and begin process execution. This requires the user to manually configure the PU’s registers (as arguments to \texttt{I\_fork\_init}),
including the heap, stack, and stack limit pointers; furthermore, it does not automatically set up the PU stack variables used by functions such as `ix_malloc`, meaning that if these functions are going to be used, the user is responsible for pushing their configuration onto the PU’s stack by hand before starting it.

Once a PU is running, one additional function becomes available, `Ix_fork_again`. This function combines the functionality of the rest of the `Ix_fork*` functions; it blocks until the specified PU finishes execution, then reconfigures it and restarts it.

The higher-level approach provides two functions, `Ix_start_multi_pu` and `Ix_start_pu`. As before, `Ix_start_mode` is used to activate mixed or multicore mode; once this is done, however, the `Ix_start_*` functions completely replace the `Ix_fork*` functions. `Ix_start_pu` is used to activate a single PU, with a specified DMEM location, DMEM size, and entry point; `Ix_start_multi_pu` is used to activate multiple PUs simultaneously, each one with the same entry point but with its own DMEM area. `Ix_start_multi_pu` also provides an optional means to prefill the data and instruction caches. Both of these functions also automatically push onto the PU stack the information necessary for functions such as `ix_malloc` to be used safely from the PU.

While much more convenient than `Ix_fork*`, these functions are not without their own problems. `Ix_start_multi_pu`, in particular, cannot be used to activate all of the PUs on some platforms; it returns a 32-bit bitmask indicating which PUs are active, or `(uint32)(-1)` if an error occurred. On a 128-PE system, there is thus no way to distinguish between “all 32 PUs are active” and “an error occurred”, both of which are represented as `0xFFFFFFFF`. This can be considered a flaw in design of the vendor library.

Furthermore, *all* of these functions — `Ix_start*pu` and `Ix_fork*` — can be called
without first calling \texttt{Ix\_start\_mode}. This will even mostly work — the PUs will activate, and code will run on them — but \texttt{Ix\_start\_mode} is responsible for more than just activating the PUs; it is also responsible for saving and restoring the PE and IMEM state when switching modes. This user error can result in hard-to-trace bugs when IMEM is not properly saved across activations.

Once processes are created, the means of managing them are the same regardless of how they were created and started. The only API functions provided are \texttt{ix\_halt} and \texttt{Ix\_join}. \texttt{Ix\_join} is given a set of PUs to watch, and blocks until all of them finish execution. \texttt{ix\_halt} is used to halt execution of the caller (or, if called by the CP, the entire chip); the caller decides whether it is \textit{finished} (in which case the \texttt{join} flag is set for the calling PU, and an \texttt{Ix\_join} watching it is permitted to return), or whether it is merely \textit{suspended} (in which case \texttt{join} is not set and the CP is responsible for either reconfiguring that PU, or resuming it).

The actual process of resuming a suspended PU must be done with the general-purpose function \texttt{ix\_spec\_write}, used for writing values to special registers. In particular, this function can be used to manipulate the \texttt{mpasn} register, which controls which PUs are active; the \texttt{mprun} register, which controls which PUs are executing; and the \texttt{mpjoin} register, which controls which PUs have the join-flag set and are thus considered to have finished execution. This is the only way to resume execution of a suspended PU, or to remotely halt the execution of a PU (i.e., without the PU itself calling \texttt{ix\_halt}).

In the AutoPilot approach, it is sufficient to utilize the \texttt{Ix\_start\_mode}, \texttt{Ix\_start\_pu}, and \texttt{Ix\_join} functions. The suspend/resume features are not exposed by the Pilot API, and the increased flexibility offered by the low-level API is not required.
3.1.2 Interprocess Communication

The primary means of interprocess communication and coordination is the *message*, a data packet sent using the C-ring that interconnects the PUs. The IMAPCAR2 message interface is extremely flexible, with three fundamental types of message (standard, synchronous, and interrupt), each one of which can be configured in various ways, in addition to some convenience functions for sending bulk data that build on those.

All messages are sent and received in a multi-stage process. First, an `ix_msg` struct is allocated (typically on the stack). Then, one of the `ix_msg_header*` functions is called to fill in the message header, configuring the message type and other parameters such as message ID and broadcast flag. The message payload, if any, is provided by assigning to struct members. Finally, one of the `ix_send*` functions is called to send the message, which blocks until the message is successfully placed on the C-ring (it does not block until the message is received). The receiver does the same, except calling one of the `ix_recv*` functions, which will block until an appropriate message arrives and its contents are recorded in the `ix_msg` struct.

Messages can be assigned a destination in two ways: message ID, or destination PU. In the former case, the message is assigned a 16-bit ID, and circles the C-ring until a PU calls one of the `ix_recv*` functions with a matching ID. In the latter case, the message stays on the C-ring until the specified destination PU calls `ix_recv*` without specifying an ID. A message can also be assigned a message ID *and* a destination PU, in which case it will stay on the C-ring until the specified PU calls `ix_recv*` and specifies a matching ID. In the case of multiple valid receivers (e.g., a message sent with an ID but no destination
PU, and two PUs waiting on the same ID), whichever one sees the message first as it circles
the C-ring will receive it (and remove it from the C-ring) — the other will remain blocked.

*Standard messages*, created using `ix_msg_header`, are the basic message type, car-
rying a payload of three `uint32` s. In addition to the destination assignment by message
ID, destination PU, or both discussed above, standard messages can also be configured as
*broadcast messages*. In this case, a set of destination PUs cannot be specified, only a count
$N$ and a message ID; the message circles the C-ring until it has been received $N$ times by
PUs calling `ix_recv*` with a matching ID. Broadcasting is part of a programming model
where the receivers effectively form a thread pool of which $N$ are required to process a
given broadcast.

*Synchronous messages*, created using `ix_msg_header_sync`, are similar to standard
messages, but elicit an automatic reply from the receiver (specifically, the contents of the
`ix_MSG` struct passed to `ix_recv*` to fill in). This is typically used when the sender needs to
know that the message has been successfully received; it can call `ix_send*` to send a mes-
sage, and immediately follow it with `ix_recv*` to wait for confirmation that its message
was received. Like standard messages, synchronous messages can be sent as broadcasts,
with the same restrictions (i.e., destination PUs cannot be specified by the sender, only a
message ID and receiver count); in this case, only one reply message is sent, from the $N$th
PU to receive the message (i.e., the one that not only receives it, but removes it from the
C-ring). These replies are produced automatically when the message is removed from the
C-ring; the receiving PU does not explicitly send them.

*Interrupt messages* are quite different. They carry no payload, have no message ID,
and cannot be explicitly received; instead, each one carries the address of an interrupt
handler, and when it reaches a PU, causes that PU to immediately save its state, execute
the interrupt handler, and then restore the saved state and resume execution. Like other
message types, interrupt messages can be broadcast, but in that case the sender has no
control over which PUs get interrupted. All it can do is specify a count, and the first count
PUs to be reached by the message will be interrupted.

Built on top of these are the \texttt{ix\_send\_msg\_blk} family of functions. These are used to
send lengthy buffers of data, automatically breaking them down into several messages that
the receiver then automatically reassembles.

The message-passing API, while initially daunting in its multiplicity and complexity
of options, is actually not that hard to use. Unfortunately, it is very easy to misuse. Sending
a message with the wrong ID or to the wrong PU will generate no warnings, it will simply
result in a message on the C-ring that will never be received — or that will be received
by the wrong process. Similarly, a size mismatch between the sender and receiver with
\texttt{ix\_send\_msg\_blk} will result in undefined behaviour at runtime.

Broadcast messages also have a few pitfalls. When sending a broadcast message, the
sender specifies the number of target PUs. However, there is no guarantee that the message
will reach that many unique PUs; rather, what is guaranteed is that it will be received
that many times before it is removed from the C-ring. This means that a single broadcast
message can be received multiple times by the same PU — which is occasionally useful for
work order dispatch, but can just as easily result in unexpected and problematic behaviour.
Worse, since \texttt{ix\_send\_msg\_blk} uses broadcast messages internally, if there are multiple
PUs waiting to receive a block message with the same ID, the message will end up split
between them!
There is no intrinsic support for message ordering, either. If multiple messages are sent sequentially, it is possible for them to be received in a different order than the order in which they were sent, even if all of them were sent from the same PU. The reason for this is that there is no internal buffering of messages; once sent, a message circles the C-ring until a PU receives it (and removes it from the C-ring). Thus, it is possible for a PU, for example, to receive the first of three messages, “miss” the second message as it is still processing the first, and then receive the third, receiving the second only on its second circuit of the C-ring. Any protocol that relies on sending multiple messages where ordering is significant must therefore handle this explicitly.

Finally, there is the possibility of \textit{C-ring flooding}. The C-ring has a limited number of slots — \( \frac{\text{PUNO}}{2} \), specifically, where \text{PUNO} is the number of available PUs — and once that many messages are on the C-ring, no more can be sent until one or more of the existing messages is received. This, unfortunately, makes it extremely easy to deadlock everyone — for example, if PUs 0-7 are all sending messages to PU 8, but PU 8 wants to send a message to the CP before it receives any of them, a deadlock will occur despite the lack of circular waiting, because — with the entire C-ring filled by the messages from PUs 0 through 7 — PU 8 will never be able to send its message, and thus, never be able to receive any of the messages already on the C-ring. A limited resource, C-ring message slots, has been exhausted, but the programmer may be unaware of this limitation, or fail to manage it properly.

As was described above, the messaging features of the IMAPCAR2 present an ordinary programmer with many choices and several hazards. AutoPilot will relieve this burden by choosing suitable methods for various use cases (e.g., payload size, broadcast
vs. point-to-point) and making them work reliably.

### 3.1.3 Global Memory Safety

Unlike the clusters that Pilot originated on, the IMAPCAR2 does have global memory. Thus, it is possible to use it for interprocess communication: what one process writes to memory, another process can read.

The primary obstacle to using this in the same manner as, for example, a pthreads-based program on a multicore desktop computer is IMAPCAR2’s lack of cache coherency hardware. Writes won’t actually propagate from a processor’s data cache to main memory until the cache fills up and cache lines begin to be flushed automatically by the hardware cache controller’s replacement algorithm; furthermore, even once the changes are committed to main memory, they won’t be visible to other processors if those processors already had the contents of that memory cached, as their caches will not be automatically updated to reflect the change.

There are two ways of coping with this (not counting “don’t use shared global memory for anything”): manual cache management, and bypassing the cache entirely.

Manual cache management is done via the `ix_cwb*` (cache writeback) and `ix_civd*` (cache invalidation) functions. The former are used to force writeback of one or more cache lines to main memory; the latter, to invalidate cache lines (so that the next read from an address previously covered by them will fetch the value currently in main memory). It seems that these make global memory useful, if somewhat awkward; Figure 3.1 demonstrates the use of these functions. In this example, a process updates a global counter by invalidating its cache entry, updating it, and then forcing a writeback.
Figure 3.1: Direct cache manipulation to update a global counter

Unfortunately, this is not as safe as it looks. The resolution of the cache manipulation functions is, at smallest, *an entire cache line* [1] — which is to say, 512 bytes. Invalidating a cache entry invalidates the entries for everything in the same cache line, and forcing a writeback writes back the entire cache line containing the value. The result of this is that Figure 3.1 is actually *extremely* unsafe. If all of the counters occupy the same cache line — which is likely — the call to `ix_civda` will clear the cached values not just of `global_counter`, but also of `my_counter`, causing the thread to lose track of its counter value. Furthermore, the call to `ix_cwba` will overwrite not just `my_counter` and `global_counter`, but also `their_counter`, potentially interfering with another thread!

The alternate solution is to avoid use of the cache entirely. 1DC provides a new storage qualifier, `outside`, which is used to signify that a value should be stored “outside the chip”; that is to say, it should reside solely in external memory and never be cached. Any access, read or write, to a variable declared `outside` will bypass the cache and access main memory directly. Figure 3.1 can then be rewritten safely as Figure 3.2, using `outside` to declare the global counter uncacheable.

This is, in general, a much better solution than manual cache management. Any variable that needs to be shared between processes can be declared `outside` and there, problem
solved! It is not, however, a silver bullet. Bypassing the cache also bypasses the cache’s advantages, which means that access to outside variables is significantly slower (3-4x slower, in fact) than access to cached variables — and the more processes are competing for access to external memory, the worse it gets. Furthermore, outside affects how the emitted code accesses the data, not how the data itself is stored, and thus — like const — can be removed by casting, and does not prevent outside variables from being passed to functions that then access them without the outside qualifier.

For protecting critical sections in code where global variables are needed, lib1dc also provides mutexes. These are fairly straightforward: ix_mutex_create creates a mutex and returns a unique ID, which can then be used to lock and unlock the mutex and, eventually, destroy it. Internally, these are managed using broadcast messages on the C-ring to coordinate between processes [39].

It is exactly because of the treacherous conditions surrounding the use of global memory described above, for which the ordinary shared-memory multicore programmer is simply unprepared, that message passing is to be preferred on the IMAPCAR2 architecture for interprocess communication. This also removes the need for mutex protection of global variables, which is a decided advantage since they have their own opportunities for misuse,
leading to data corruption or deadlocks. With Pilot style message passing, each process continues to make fast access to its cache, but the need for manual cache management is avoided.

3.1.4 Image Buffer Access

As one of the primary purposes of PU processes is to analyze regions of interest in an image, they need an efficient way of accessing the image buffer. This is provided by the `ix_roi_read` and `ix_roi_write` functions, which have two purposes: efficient cache usage, and linearization of rectangular image regions.

Efficient cache usage is done by using specialized opcodes (these functions are implemented in IMAPCAR2 assembler, not 1DC) which bypasses the normal cache line mechanism entirely. In this way, only as much cache as needed for the region of interest is used, rather than filling every cache line hit by the region.

Linearization is necessary (or, at least, desireable) because in most cases, the lines of a region of interest will not be contiguous in memory. Images are stored as lines packed end to end, so the first byte of the image is the upper left pixel and the last byte is the lower right. This means, however, that if the image is 640 pixels wide and the region of interest is only 40, there will be a gap of 600 pixels (which is not necessarily 600 bytes, depending on the bit depth of the image) between the end of one row of the region of interest and the start of the next. Figure 3.3 illustrates this.

This is an area where AutoPilot does not need to get involved. The SDK’s ROI functions are straightforward for the programmer to call directly, and there is no natural mapping of ROI manipulation into Pilot’s process/channel abstractions. The main use of Au-
Figure 3.3: Region of interest access, with and without ix_roi_* functions [42]

toPilot in this context would be for one process to pass the address of an ROI to another
process via a channel message, as, for example, the master distributes data to workers, or
as workers passes data along a pipeline of tasks.

3.2 The pthreads Library

In addition to the core functions provided by lib1dc, the IMAPCAR2 comes with an
implementation of the POSIX threading API, pthreads [23]. This is a standard API for
threaded programming, in common use on UNIX and Linux based systems and with ports
to other platforms. However, it was originally designed for programming on desktops and
servers, not embedded systems — which is to say, on cache-coherent systems with shared
memory. As such, the original API focuses primarily on ways to safely manage shared
memory via synchronization primitives such as mutexes.

The IMAPCAR2 version of pthreads implements a subset of pthreads, omitting some
functions, and additionally adds a number of non-POSIX-standard functions to the pthread_
namespace to expose various hardware features of the chip that would otherwise need to be
accessed via the lib1dc API. These are detailed below.

54
The library only supports the use of the pthreads synchronization and communication functions for communication between threads. The manual warns users not to use global memory for communication or, if global memory must be used, only ever declare it outside so that it bypasses the cache. Similarly, the use of IMAPCAR2 message passing functions in any way except via the pthreads API is forbidden, as it interferes with the operation of pthreads itself.

 pthreads mode is initiated by calling \texttt{pthread\_start\_np} and passing it a function to execute. In contrast to AutoPilot and the lib1dc API, both of which have calls to switch modes that return as soon as the mode switch completes, all pthread processing is contained within this call. The function passed to it becomes the “main thread” function, executing on the CP, and is responsible for creating new threads; \texttt{pthread\_start\_np} does not return until all created threads have exited, at which point it also automatically returns the system to SIMD mode.

### 3.2.1 Thread Management

As in normal pthreads, threads are created with \texttt{pthread\_create}, waited for and freed with \texttt{pthread\_join}, detached with \texttt{pthread\_detach}, and exited with \texttt{pthread\_exit}. Like normal pthreads, IMAPCAR2 pthreads decouples threads from processors; the user can create as many, or as few, threads as they wish. On startup, the library spawns a separate process, the thread manager, which is responsible for automatically assigning threads to processors; it is also capable of saving the state of a thread and resuming it later (possibly on a different processor), which is used to ensure that all threads get some amount of execution time in circumstances where there are more threads than available PUs.
The manager thread is also responsible for keeping track of thread and processor state, mutexes, and condition variables, and it does so by using broadcast messages on the C-ring. This is the reason for the restriction on message passing by the user: user-created messages may interfere with the manager thread, or threads waiting for a reply from the manager thread, or may even lead to a C-ring flood.

For managing threads more precisely, a number of extension functions not specified by the POSIX pthreads standard are provided. Specifically, there are a number of `pthread_attr_*_np` functions which can be used to get or set various nonstandard thread attributes — the size of the EMEM and IMEM stacks reserved for that thread when executing on the CP, and thread’s processor affinity. It is also possible to get the processor ID of the processor that the manager thread is running on.

A key difference between pthreads and the Pilot approach is that users can only create as many processes are there are available processors; they cannot oversubscribe them as can be done with pthreads. The Pilot approach is compatible with scalable parallel program organization, and eliminates thread scheduling overhead.

3.2.2 Synchronization

Not all of the standard pthread synchronization primitives are available. In particular, the optional barrier and spinwait APIs are not implemented; nor are semaphores, which — while not part of the pthreads standard — are part of POSIX.1 and are traditionally provided by libpthreads on systems supporting them. Only mutexes (in the `pthread_mutex_*` namespace) and condition variables (in the `pthread_cond_*` namespace) are supported.

In addition to this, IMAPCAR2 pthreads provides two `pthread_msg_*_np` functions
for sending and receiving messages via the C-ring. Messages are considered to be opaque byte buffers, and are sent by copying them into main memory (with cache writeback) and then sending a pointer to the destination thread. It is up to user code to translate C data types into byte buffers and back again (for example, by casting to and from struct pointers). These functions avoid C-ring flooding by communicating with the manager thread to ensure that both sender and receiver are ready before actually transferring data.

An advantage of message-passing programming is that messages provide both inter-process communication and synchronization. Thus, there is no need for synchronization features such as condition variables. The effect of a barrier can be supplied by a globally synchronizing communication.

### 3.2.3 Other IMAPCAR2-Specific Extensions

In addition to the functions described above, IMAPCAR2 pthreads provides a few extra utility functions. `pthread_sleep_np` is used to temporarily suspend a thread (potentially descheduling it and letting another thread run on the processor it was previously occupying). `pthread_time_np` is used for timekeeping, and `pthread_timer_cp` for overriding how the pthreads library keeps track of time. Finally, there are five functions for transferring data between IMEM and EMEM — two (`pthread_ememrw*_np`) for high and low priority synchronous transfers, two (`pthread_ememrw*_wno_np`) for high and low priority asynchronous transfers, and one (`pthread_wwait_np`) to wait for asynchronous transfers to complete.

In AutoPilot, there is no equivalent for the IMEM/EMEM transfer functions or for `pthread_sleep_np`. The former are thin wrappers around the lib1dc API that could easily
be replicated if needed, but they have no equivalent in the standard Pilot API and were thus deemed not suitable for inclusion in this prototype. The latter is used to mark a thread “inactive” so that the pthreads manager thread can suspend it and schedule a different thread on that core; as AutoPilot does not allow overcommitting, this is not a useful capability.

As for timekeeping, this is supplied by the PI_StartTime and PI_EndTime functions. The former starts a timer (or resets an already running one); the latter resets the timer and returns the time elapsed since the last call to either function. These functions internally rely on IxPttime, which accesses a timing register only available on the CP and thus cannot be called from PUs.

### 3.3 AutoPilot Design

AutoPilot’s API closely mimics that of standard Pilot, with some changes to support the IMAPCAR2’s specific capabilities and limitations. The process/channel/bundle architecture is intact, as is the division between configure and execution phases; existing Pilot programmers will find it familiar despite the changes.

The primary changes are the relaxation of bundle differentiation as to usage, absence of PI_Reduce (due to lack of hardware or API support), and support for multiple configure-execution-shutdown cycles in a program’s lifetime. There is also one new function, PI_CreateDefaultProcesses, for automatically setting up the common case where the user wants one process on each PU, with channels to and from each one.
3.3.1 PI_Configure

Unlike normal Pilot programs — which start up with multiple processes, and exit completely when processing is completed — AutoPilot programs on the IMAPCAR2 may switch between SIMD and MIMD multiple times, typically once (or more) per frame. This means that the process of starting up the various nodes — which, in normal Pilot, is handled by mpirun before Pilot gets control — must instead be handled by AutoPilot itself. In effect, the caller needs to be able to tell PI_Configure how many nodes are needed, and PI_Configure needs to respond by switching the chip into MIMD mode and activating at least that many PUs (or failing sensibly if that is not possible, for example, if the caller has requested more PUs than this model of IMAPCAR2 has available). It also means that PI_Configure needs to be callable multiple times per program run — it cannot safely assume that PI_Configure, PI_StartAll, and PI_StopMain are only ever called once.

AutoPilot also requires additional information from the user. Whereas Pilot programs generally assume that each process can allocate memory at will, in AutoPilot each PU must be assigned a data memory region when activated, and all stack and heap operations conducted by that PU must remain inside that region. This means that AutoPilot needs to know up front how much memory to assign to each PU.

This is accomplished by adding two arguments to PI_Configure: \textit{dmensize} and \textit{dmemptr}. \textit{dmensize} specifies the amount of memory that should be allotted to each PU; \textit{dmemptr} is a pointer to the start of a contiguous region of memory that AutoPilot is permitted to use for DMEM. If \textit{dmemptr} is not specified, AutoPilot automatically allocates DMEM for each process from the CP heap when PI_Configure is called, and frees it
when the system returns to SIMD mode. Specifying it permits the caller to allocate memory once, at startup or statically, and re-use it for each call to `PI_Configure`, obviating the need for AutoPilot to perform multiple heap allocations and frees every time it enters and leaves MP mode.

The IMAPCAR2’s paged memory model makes AutoPilot initialization potentially even more complicated. In “normal” operation, each PU is assigned data memory residing in the lowermost page — thus sharing it with other PUs, the CP, and compiler-allocated global and static data. This is the simplest operating mode, allowing M-ring communication between PUs using `ix_memcpy` and requiring no special work to access globals.

It is also possible, however, to assign each PU a data memory region residing in an upper memory page. In this case, the PU cannot access *anything* outside that page except by using functions that bypass normal memory access, such as `ix_roi_read`. This not only means that PUs can’t directly access on another’s data memory (a technique that AutoPilot uses for transferring very large messages), but also means that the PU cannot access globals at all, unless those globals are first copied into the page its data memory occupies (in which case any changes it makes to those globals won’t be reflected in other pages).

At present, AutoPilot does not support placing PUs outside page 0, due to such issues. Section 6.4 discusses possibilities for future support for this feature.

### 3.3.2 Configuration Phase Functions

The remainder of the configuration phase is very much like normal Pilot. User code calls `PI_Create*` (and possibly `PI_CopyChannels`) to create processes, channels, and bundles. Once everything needed has been created, `PI_StartAll` is called to begin ex-
execution. All of the normal Pilot functions are available; there are, however, some minor differences.

The first and most obvious is that the PI_Create* functions do not return pointers, but PIPROCESS, PI_CHANNEL, and PI_BUNDLE values directly. These are actually type-aliased to int32, being opaque handles into AutoPilot with some commonly-used information encoded in the handle itself. There are a number of reasons why AutoPilot API functions do not return pointers. Early revisions of the design, working on the assumption that AutoPilot would need to support placing PU DMEM in upper memory pages — and predating the solution to upper page DMEM discussed in Section 6.4 — avoided the use of pointers as much as possible, to minimize situations where a process mapped into one page would need to dereference a pointer to data in another page. Later versions dropped support for cross-page operation in the prototype implementation; the pointerless design was kept to support future implementation of this feature. The current design for cross-page operation support does not mandate a pointerless design, but with the pointerless version already implemented and working, it was decided not to rework the prototype to use pointers.

The second is that PI_CreateBundle no longer distinguishes between select, broadcast, scatter, gather, and reduce bundles; bundles are simply classified as reader bundles or writer bundles, based on whether the common endpoint is a reader or writer of the channels included in the bundle. The reason for Pilot’s strict bundle differentiation was that the process at the (logically) non-collective end had to know which collective MPI operation to invoke. For example, each recipient of a broadcast, though ostensibly reading from a single channel, nonetheless has to call MPI_Bcast just like the broadcasting process. Furthermore, this requirement meant that once a channel had been incorporated into
a broadcast-usage bundle, it could not also be used for individual writes, as the receiving process would not know when to call `MPI_Recv` instead of `MPI_Bcast`.

The underlying IMAPCAR2 API can be viewed as more elementary than MPI, in that it largely lacks sophisticated collective operations. The positive consequence is that AutoPilot need not maintain Pilot’s artificial bundle differentiation, and it can also offer the flexibility of reading and writing on channels that have been incorporated into bundles. This in turn cuts down the need to define multiple channels between the same pair of processes. Thus, a single bundle can be freely used for any operation its directionality supports — broadcast and scatter for writer bundles, and select and gather for readers.

Apart from these two changes, and the changes to `PI_Configure` itself discussed earlier, the configuration phase behaves identically to normal Pilot. This was not always the case; earlier versions of the design considered two other possibilities for the configuration phase, both of which differed more dramatically from Pilot.

The first design considered would have split the entire “configuration phase” into a separate, user-provided function. In Pilot, the same program starts up simultaneously on all nodes; each running instance then executes the entire configuration phase. The construction of process, channel, and bundle information is fully deterministic, guaranteeing that Pilot’s data structures in all nodes will agree regardless of their respective processor architectures. Once `PI_StartAll` is called, each node decides which process it “really” is based on its MPI rank, and begins execution of the corresponding Pilot process — with one node, `PI_MAIN`, executing none of them and instead returning from `PI_StartAll`. This design was intended to replicate that behaviour: `PI_Configure` would be passed a configuration function, which would then execute simultaneously on all of the PUs (and the CP).
Returning from the configuration function would be an implicit \texttt{PI\_StartAll}; on the CP that would return from \texttt{PI\_Configure}, and on the PUs, it would begin executing the user function. After investigation, it was concluded that there was no advantage in implementing configuration this way, as opposed to running configuration once on the CP and then broadcasting the results to the PUs.

The second possibility was, in effect, doing away with \texttt{PI\_MAIN} entirely. In this design, \texttt{PI\_StartAll} would start all of the PUs and then block; when all PUs finished executing, it would return either \texttt{PI\_SUCCESS} or an error code — depending on whether everything went fine, or whether an error occurred and \texttt{PI\_Abort} had to be called. This design was motivated primarily by error handling considerations; it would have made it easier for \texttt{PI\_MAIN} to detect errors at runtime and handle them gracefully.

The drawbacks, however, were severe. With the CP blocked, the programmer loses an entire potential process — the most powerful one, as the CP has larger caches than the PUs. One also loses any ability to make use of the SIMD array in mixed mode. Furthermore, upon further correspondence with S. Kyo, it was determined that there was no need for it; the expected error handling model for the IMAPCAR2 is that errors will be reported to the chip’s controller (the debugger in a development environment, or the CPU in production), and it is then the responsibility of the controller to collect any information needed and restart the IMAPCAR2. This is an error reporting model that can be easily handled with interrupt messages, leaving the CP free to act as \texttt{PI\_MAIN} and do useful work as the master coordinating multiple worker processes.
3.3.3 Execution Phase

Once \texttt{PI\_StartAll} is called, the normal execution-phase functions can be used. \texttt{PI\_Read} and \texttt{PI\_Write} behave equivalently to the Pilot versions, with some changes to the format strings; the AutoPilot formats — and their corresponding Pilot ones, where applicable — are detailed in Table 3.1. In particular, \%m is no longer supported (as there is no “MPI” datatype), and \%p is added as a “physical pointer” type — in effect a \texttt{uint32} recording a (page-independent) address in physical memory, for use with functions like \texttt{ix\_roi\_read}. The three-letter long formats are also abbreviated to two characters, to work better with the 16-bit data registers of the IMAPCAR2.

In AutoPilot, some of the formats are redundant. For example, there is no possibility that messages will need to be passed between architectures with different formats for signed integers, meaning that the signed and unsigned formats can be used interchangeably. Similarly, as there is no character set conversion needed between processors, the \%c (char) and \%b (unsigned byte) formats are functionally equivalent. These redundancies are permitted for two reasons - to keep code compatibility with the Pilot API on architectures that \textit{do} require these different formats, and to make it easier for the programmer to clearly express their intent in the format string.

Collective operations deserve special mention. The IMAPCAR2 has specialized hardware that can be used to implement some of them efficiently; in particular, there is a ’broadcast’ C-ring message type that works well with \texttt{PI\_Broadcast} and \texttt{PI\_Scatter}. On the other hand, there is no hardware support for optimization of reduce/fold operations. While reduce operations could be implemented in the future if needed, AutoPilot does not support
<table>
<thead>
<tr>
<th>Pilot</th>
<th>AutoPilot</th>
<th>IMAPCAR2 type</th>
<th>Standard C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d, %i</td>
<td>%d, %i</td>
<td>int16</td>
<td>int</td>
</tr>
<tr>
<td>%ld, %li</td>
<td>%ld, %li</td>
<td>int32</td>
<td>long int</td>
</tr>
<tr>
<td>%lld, %lli</td>
<td>%lld, %lli</td>
<td>int64</td>
<td>long long int</td>
</tr>
<tr>
<td>%u</td>
<td>%u</td>
<td>uint16</td>
<td>unsigned int</td>
</tr>
<tr>
<td>%lu</td>
<td>%lu</td>
<td>uint32</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>%llu</td>
<td>%llu</td>
<td>uint64</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>%b</td>
<td>%b</td>
<td>uint8</td>
<td>unsigned char</td>
</tr>
<tr>
<td>%c</td>
<td>NA</td>
<td>NA</td>
<td>char, with character set conversion</td>
</tr>
<tr>
<td>NA</td>
<td>%c</td>
<td>char</td>
<td>char, no character set conversion</td>
</tr>
<tr>
<td>%f</td>
<td>%f</td>
<td>float</td>
<td>float (single precision)</td>
</tr>
<tr>
<td>%lf</td>
<td>%lf</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>%llf</td>
<td>NA</td>
<td>NA</td>
<td>long double (“quad”)</td>
</tr>
<tr>
<td>%m</td>
<td>NA</td>
<td>NA</td>
<td>MPI_Datatype, for user defined types</td>
</tr>
<tr>
<td>NA</td>
<td>%p</td>
<td>emem_ptr_t</td>
<td>unsigned long int, EMEM address</td>
</tr>
</tbody>
</table>

Table 3.1: Format strings in AutoPilot
it in the current implementation; application code that requires it can easily implement it in the meantime using a loop.

There is also no hardware support for PI_Select and its sister functions, PI_TrySelect and PI_ChannelHasData; the IMAPCAR2 offers no way of checking for the presence of a message on the C-ring other than actually receiving that message. Fortunately, the groundwork for these functions is already laid in the implementation of C-ring flood avoidance, and AutoPilot provides them.

Support was planned for two collective operations not present in normal Pilot, which the IMAPCAR2 has hardware support for: distribute, in which one or more messages are sent to whichever PUs (in a set of bundle readers) are ready to receive them, and funnel, efficiently combining PI_Select, PI_GetBundleChannel, and PI_Read into a single call which efficiently reads the next incoming message on any channel in a bundle. They were not implemented due to time constraints, but would not be difficult to add, and are discussed further in Section 6.2. If they had been, distribute was planned to be merged with PI_Broadcast, which would accept an additional count argument; this would determine how many copies of the message to distribute, or, if the special value PI_BROADCAST_ALL, would behave like a normal broadcast.

3.3.4 Error Handling

Normal Pilot has two approaches to error handling, return or abort, which can be configured at runtime by setting the global flag PI_OnErrorReturn. In abort mode, the default, all errors are considered fatal. An error will cause Pilot to call PI_Abort (which can also be called at any time by user code), immediately terminating the program on all
nodes with an informative error message.

In *return* mode, all error handling is delegated to the caller; any Pilot function that encounters an error returns an error code, and the caller can then get more information from the `PI_Errno` and `PI_ErrorText` variables. This mode is included for automated testing, and is not normally used in program code.

AutoPilot currently only supports `PI_Abort`, with some configuration possible via a preprocessor option: if `PI_DEBUG` is 0, `PI_Abort` remains available for user code but all error checks by AutoPilot are skipped (improving performance slightly). If it is 1 — the default — Pilot will perform internal error checks as normal and will call `PI_Abort` if an error occurs. If it is 2, AutoPilot will additionally log all calls from user code, allowing `PI_Abort` to report the exact file and line the AutoPilot function reporting the error was called from. (Values above 2 enable debug logging, which greatly degrades performance but is useful for tracking down bugs in AutoPilot itself.)

In normal use, `PI_DEBUG` is set to 1, enabling basic error checking; in a typical embedded deployment, the IMAPCAR2 will be subordinate to an external monitor that can log errors and reset the IMAPCAR2. A setting of 0 can be used for a modest performance boost, but without error checking, some classes of errors will result in deadlocks; in that case, the monitor must be capable of detecting this and resetting the chip when it occurs.

### 3.3.5 Shutdown

`PI_StopMain` is effectively unchanged from normal Pilot except that it does not require an status code. Unlike a normal program, it is not assumed that `PI_StopMain` indicates the end of the program; rather, it indicates that MIMD processing is complete (or will
be complete once the PUs finish) and that the system should return to SIMD mode. Thus, 
\texttt{PI\_StopMain} blocks until all of the PUs have finished, then shuts them down in good or-
der and returns. At this point the program is free to do whatever it wants with the collected 
results of the MIMD processing, and can call \texttt{PI\_Configure} again to switch the program 
back into MIMD mode at a later time.
Chapter 4

Implementation

This chapter contains details on how AutoPilot is implemented. The entire library is written in the subset of 1DC that is permitted for MIMD programming, meaning that outside data is available, but sep and the various SIMD operators and keywords are not. The library was developed primarily on Linux, using Wine [13] to run the (Windows-based) development tools; for source-level debugging and access to the test board, a Windows virtual machine running the Eclipse-based development environment was used.

Section 4.1 details switching into (PI_Configure) and out of (PI_StopMain) Pilot mode; Section 4.2 covers the configuration phase and Pilot data types; Section 4.3 and Section 4.4 cover point-to-point and collective operations respectively; and Section 4.5 covers the error handling mechanisms.

As a general policy, AutoPilot tries to avoid cache writeback and invalidation. The problems detailed in Subsection 3.1.3 — including cache staleness and overlapping writebacks — apply to AutoPilot as well. As much as possible, it uses outside variables and message passing to avoid needing to share access to cached data. In cases where this is not possible (for example, when the user has passed a lengthy buffer to PI_Write), it prefers to use the cache-to-memory and memory-to-cache versions of memcpy, ix_memcpy_c2nc and ix_memcpy_nc2c, which allow specification of precise extents for copying and thus do
not suffer from the same dangers as direct cache manipulation, at the cost of performance.

4.1 Mode Switching

Mode switching occurs when the user initiates Pilot mode (by calling \texttt{PI\_Configure}) or leaves it (by the CP calling \texttt{PI\_StopMain} and all other processes exiting).

Entering Pilot mode is complicated. The user provides a desired \textit{count} of active PUs, a \textit{dmensize} indicating how much DMEM each should have available, and an optional \textit{dmemptr} pointing to a preallocated DMEM region. AutoPilot then calls \texttt{Ix\_start\_mode} (selecting either mixed mode or MP mode depending on how many PUs have been requested) and loops over the PUs, calling \texttt{Ix\_start\_pu} on each one, until \textit{count} PUs have been activated. If a \textit{dmemptr} was provided, it is partitioned into a DMEM region for each PU; if not, AutoPilot calls \texttt{ix\_malloc} to allocate memory for each PU (and records the pointer in an array so that it can be freed when the system leaves Pilot mode). PUs are initialized with a small function that initially does nothing but wait for configuration information (which it will receive when \texttt{PI\_CreateProcess} is called).

PU DMEM areas, additionally, are required to be aligned to a 1KB (1024) boundary, and have a size a multiple of 1024 bytes. When AutoPilot is allocating DMEM, this requirement is automatically satisfied by \texttt{ix\_malloc}. When the user is providing AutoPilot with a memory region to use, however, it is not always possible to satisfy this alignment. In that case, AutoPilot will actually seek into the buffer for the next 1KB boundary and, for that reason, the user needs to provide a buffer to AutoPilot that is 1KB larger than \textit{count} * \textit{dmensize}. In either case, the stack and heap information and PU registers are auto-
matically configured by \texttt{Ix\_start\_pu}.

The actual size required for the DMEM region depends on the algorithm being executed. Due to the alignment and size constraints, the smallest possible DMEM size is 1KB. In practice, 4KB is a sensible “default minimum”, providing a few kilobytes for stack and heap operations while still fitting entirely inside the PU’s data cache. The upper bound on PU DMEM size is determined by the amount of space available in CP DMEM to allocate it from, which means it can be 512KB or more if CP memory is used sparingly; the vendor literature, however, recommends keeping PU DMEM regions to 8KB or less to make it easier to fit the entire working set in cache [1].

Returning from Pilot mode to SIMD mode is much simpler. \texttt{PI\_StopMain} calls \texttt{Ix\_join} to block until all of the PUs have finished executing. Once this occurs, it checks the DMEM array to see if any allocated memory needs to be freed, and if so, frees it. Finally, it calls \texttt{Ix\_start\_mode(IX\_MODE\_SIMD)} to return to SIMD mode.

4.2 The Configuration Phase

During this phase, the application creates processes, channels, and bundles, and AutoPilot builds its internal tables describing them. Some data is used only within the configuration phase, while others must persist into the execution phase.

\texttt{PI\_PROCESS} values are an example of the first sort. \texttt{PI\_CreateProcess} configures the next available PU (by sending it a message containing the work function, index value, and userdata pointer) and then returns that PU’s ID cast to a \texttt{PI\_PROCESS}. \texttt{PI\_PROCESS} values are used only for designating channel endpoints during configuration, and are not
needed thereafter.

Channel data structures, on the other hand, must persist into the execution phase. Each channel has an origin and destination process and an associated tag (so that multiple channels connecting the same two processes in the same direction can be distinguished). Furthermore, each channel has two variables associated with it: has_data is true if the channel is ready for reading, and bundle_tag is nonzero if the channel belongs to a bundle (in which case it is the bundle’s associated tag). Tags are 16-bit values corresponding to IMAPCAR2 message IDs (which have no predefined meaning; they are completely arbitrary and can be chosen by the user — or in this case, chosen by AutoPilot on the user’s behalf — using any scheme desired); AutoPilot starts at 0 and simply counts upwards. The bundle_tag and has_data variables are stored in outside arrays, thus guaranteeing any PU using them always has an up to date view of them (at the cost of losing the cache’s performance benefits when accessing them). The origin, destination, and tag — being fixed at channel creation time — are packed into the four bytes of a uint32, which PI_CHANNEL is a typedef for.

Bundles are even more complicated. Each bundle has a directionality, a tag, a common endpoint, and two or more non-common endpoints. The directionality, tag, and common endpoint can all be stored in the PI_BUNDLE itself in the same manner as a PI_CHANNEL, but the other endpoints need to be stored somewhere, and are required during the execution phase for some operations. This is accomplished by storing (in a separate array, indexed by bundle tag) a pointer to the PI_CHANNEL array passed to PI_CreateBundle — to avoid allocations, this memory is used as is rather than copied.

The scope of these values was a problematic subject during the design of AutoPilot.
In standard Pilot, the \texttt{PI\_PROCESS} pointers must be available throughout the configuration phase, but are not subsequently necessary; the \texttt{PI\_CHANNEL} and \texttt{PI\_BUNDLE} values, in contrast, must be in scope both to the configuration phase code (which creates them) and the execution phase code (which uses them to send and receive messages). This is not a problem, despite the lack of shared memory, because the configuration code executes on every node in parallel, and thus each node gets an identical copy of these values and the internal tables backing them.

On the IMAPCAR2, in contrast, this is not automatically the case. The worker functions still need to be able to refer to \texttt{PI\_CHANNEL} and \texttt{PI\_BUNDLE} values during the execution phase, but the configuration phase code may not be executing on all nodes. Some mechanism, then, is needed for these values to be in sync across all nodes — ideally without drastically changing the API from normal Pilot. One implementation, discussed in Subsection 3.3.2, would have behaved in this manner, but it was ultimately not used.

Other early designs were subject to the constraint that AutoPilot had to be able to allocate PU DMEM in upper memory pages, which meant that process and channel information couldn’t be safely stored in global scope if the worker functions executing on the PUs were to have access to it. The proposed solution to this was to use only the addresses of the process and channel variables, not their values; channels would be created, for example, by calling \texttt{void PI\_CreateChannel(PI\_CHANNEL * c, PI\_PROCESS * from, PI\_PROCESS * to)}. Application code would be required to allocate the \texttt{PI\_CHANNEL} variables in global storage, but would pass pointers to them using the C address-of operator; AutoPilot would then never dereference the pointer, but use the address it referred to as a lookup key for that channel. Upon entering the execution phase, the tables of these
pointers would be transmitted to the PUs using the C-ring.

The design ultimately settled on was made possible by the decision that the initial prototype did not need to support upper memory pages, and the discovery of better options for implementing upper memory DMEM once that support is added. In this design, the channel and bundle values are simply declared in global scope in such a manner that they are in scope to both the configuration phase and execution phase functions, as in normal Pilot. These values are designed to be immutable: once the configuration phase completes, none of these channel or bundle values will change. Thus, there is no possibility of cache decoherency for them during the execution phase. To ensure that all processes have the complete set of channel and bundles available, the CP does a full cache writeback before the worker functions are called, and thus, before the PUs perform any data memory reads or writes.

Once the configuration phase is done, the user calls `PI_StartAll` to start the processes running. In AutoPilot, this first performs a cache writeback to ensure that all PUs have access to up to date values for bundle and channel data, and then sends a broadcast message, with the number of receipts equal to the number of active PUs. The message tag is a special tag (reserved by AutoPilot), and is used both for this broadcast, and for the earlier configuration messages sent by `PI_CreateProcess`. This means that every active PU, whether it has been assigned a process by `PI_CreateProcess` or not, will receive this message; the ones that have not been configured will react by shutting down with `ix_halt`, and the ones that have will call their configured work function, and call `ix_halt` only after this function returns.

As soon as the broadcast message is successfully sent, `PI_StartAll` returns and the
4.3 Read and Write

Channel reading and writing is, regrettably, not as straightforward as it appears. In principle, one should simply be able to have the reader and writer unpack the `PI_CHANNEL` to determine the tag, origin, and destination; the writer calls `ix_send_msg`, the reader calls `ix_recv_msg`; and the message is sent and received with no fuss whatsoever. Early builds of AutoPilot, in fact, did exactly this.

Unfortunately, in practice, this is problematic. Specifically, the C-ring flooding issue discussed in Subsection 3.1.2 poses a problem: if the number of AutoPilot processes exceeds the number of available C-ring slots, the system is vulnerable to deadlocking due to C-ring flood even in designs that, in theory, should be deadlock-free due a lack of circular waiting.

A number of possible solutions were considered for this. The most straightforward (from a shared memory perspective) is simply to associate a pair of flags with each channel, one indicating that the channel is ready for read and one indicating that it is ready to write, and not send any messages until both of these flags are set. The problem this creates, however, is that `PI_Read` and `PI_Write` must now poll these flags — which means that the processes that set them must set them and then perform a cache writeback to ensure the new value makes it into main memory, and the processes that poll them must repeatedly invalidate the corresponding cache line to ensure that they are seeing the most recent version of the flag. In addition to violating the recommendation that as much data be kept
in cache as possible, this also opens the door to the coherency failures due to overlapping writebacks discussed in Subsection 3.1.3.

Another design considered was to have a “manager process” akin to the one used by the pthreads library, which would maintain a table of channels and their write-ready and read-ready flags. Processes wanting to perform communication would first send a message to the manager and wait for a reply; the manager would send that reply only when it was safe to communicate. There is still a C-ring flood possibility here — if the manager is attempting to send a reply, but \( \frac{PUNO}{2} \) processes are all trying to send messages to it and have filled the C-ring — but this can be avoided by never using calls that block indefinitely, and instead letting the manager alternate between waiting to send and waiting to receive.

The main disadvantage of this approach is that it requires an entire process to be dedicated to channel management, and also requires calls such as \( \text{PI\_ChannelHasData} \) to use a message pass to query this manager. It also bottlenecks all communication on the manager; a large number of independently communicating processes must all wait until the manager is ready to communicate with them, even if they are independent from each other.

A variant on the previous design would have been interrupt-based, with the CP maintaining the channel tables and individual processes sending interrupts to it to invoke the management functions. This would not require a dedicated manager process, but would exacerbate the bottlenecking issues: since interrupts cannot carry a payload, each attempted read or write would now require two messages — one to invoke the interrupt handler, and another carrying the payload — and context switching would add more overhead still.

The solution that was actually implemented is a refinement of the last, interrupt-based approach in which each process, in effect, acts as its own manager for all channels that ter-
Figure 4.1: Sequence Diagram: Read/Write (Reader First)

When `PI_Write` is called, the writer sends an interrupt message to the reader, immediately followed by a normal “ready to send” (RTS) message containing information about the attempted write (such as which channel is being written to). The interrupt message does not require the reader to be ready for it; the interrupt handler will be invoked regardless. The interrupt handler on the reader receives the RTS message, guaranteeing that it doesn’t remain on the C-ring indefinitely, and sets the `has_data` flag on the channel — a flag which, since it is only ever written or read by the process reading from the channel, does not need to be synchronized with main memory or other processes, but can be stored entirely in cache. The writer then waits for a “ready to receive” (RTR) message from the reader.

The reader, in turn, busy-waits until the `has_data` flag is set by this interrupt (see
Figure 4.1); if the reader calls \texttt{PI\_Read} after the writer calls \texttt{PI\_Write}, it will already have been set by the interrupt handler (Figure 4.2). Once this flag is set, the reader sends the RTR message to the writer. This message does not just indicate that it is ready to receive; it also contains a pointer to the results of parsing the format string passed to \texttt{PI\_Read}. The writer receives this message and compares the two format strings to ensure that there are no type mismatches before sending the data. This procedure thus becomes part of AutoPilot’s mechanism for detecting erroneous use by application code and exposing bugs.

This is considerably more expensive than just jumping right in and sending the message, and there may be some room for improving performance, but it is necessary to avoid the possibility of C-ring floods. Indeed, a sample program provided by the vendor was found to deadlock from C-ring flooding in certain circumstances (see Section 5.3), show-
ing that even programmers experienced with the native API can fall prey to this hazard.

The actual data transmission occurs once this handshaking and format string comparison procedure is complete. The writer sends a single C-ring message for each data item — each format code — passed to `PI_Write`. In cases where the message consists of multiple values (i.e., the format string contains multiple codes, such as "%d,%d", each one is sent in its own C-ring message. Each message carries a sequence number, as it is possible (as discussed in Subsection 3.1.2, and demonstrated in early builds of AutoPilot demonstrated) to receive the messages out of order. The sequence number allows the receiver to ensure each message is matched to the correct data item.

In cases where a data item is not a single value but an array of values — for example, the format "%32u" — the data itself is not sent in the C-ring message. As C-ring messages have a limit of 12 bytes, some of which is already used by sequence information, large data items must be either split across multiple messages or sent in a manner that bypasses the C-ring. AutoPilot uses the same technique for large messages that pthreads uses for all messages: the writer calls `ix_memcpy_c2nc` to ensure that the contents of the array are written back to main memory. It then sends a pointer to the array to the reader, which copies the data into the destination buffer using the inverse operation, `ix_memcpy_nc2c`.

### 4.4 Broadcast and Select

Two collective operations are supported by AutoPilot, *Broadcast* (on writer bundles) and *Select* (on reader bundles). Broadcast is used to send a copy of the same message to every process reading from a bundle; Select, to find a channel in the bundle that is ready
for reading.

The implementation of Select, via PI_Select and PI_TrySelect, is straightforward. PI_Select repeatedly scans every channel in the bundle, checking the has_data flag on each one; as soon as it finds one with the flag set, it returns that channel. Between scans, it uses ix_nopwait to briefly wait for channels to become ready. There is no efficient way for it to “sleep” until a channel becomes ready for reading; while ix_halt can be used to suspend execution of the PU, it cannot then be resumed except by the CP. Unlike a desktop system, however, a busy-waiting process isn’t occupying a CPU that some other task could be scheduled on — each PU is either running the specific PI_PROCESS assigned to it, or is halted.

PI_TrySelect is essentially the same, except that if it does not find a ready channel in its first scan of the bundle, it returns rather than starting a new scan.

PI_Broadcast is complicated. First, the sender needs to mark all of the channels involved as ready for reading, as part of the C-ring flood avoidance handshake. Ideally, this would be accomplished by broadcasting a single message. Marking a channel ready is done using an interrupt message, however, and while interrupts can be broadcast, doing so is not reliable. The sender has no control whatsoever over which PUs are interrupted — in practice, it will be the first $n$ PUs which do not have interrupts masked. So, it first needs to iterate over the entire channel list for the bundle and send a separate interrupt for each one.

As each PU is interrupted, the broadcaster follows the interrupt with another message indicating which channel should be flagged ready for reading, in the same manner as PI_Write. In an ideal world, the sender would interrupt every PU, and then send a single broadcast message to all of them. Unfortunately, this is not possible due to the way in-
interrupts and broadcast messages work. An interrupted PU receives no information except for the fact of the interrupt itself; in particular, there is no way to pass arguments to the interrupt handler, and it has no way of telling which PU interrupted it. This means that the “channel ready” interrupt handler, when listening for that handshake-initiating message, must do so by listening for any message sent to that PU that uses the tag reserved for such messages.\textsuperscript{1} The upshot of this is that it is up to the sender of the ready-message to target the message at the PU that needs to receive it, and in the case of broadcasts, this is not possible — which means that a broadcast ready-message will potentially be picked up by other PUs, prompting them to mark as ready a channel they are not actually connected to (and leaving the message intended for them on the C-ring forever, which — if this happens enough times — will eventually lead to a deadlock). Consequently, the sender must also send the ready-messages individually.

Once these two messages — interrupt and ready — have been sent, it’s time for the reader to reply with format string information, in the same manner as when \texttt{PI\_Write} instead of \texttt{PI\_Broadcast} is being used. Again, there’s no easy way to do this en masse; it would be nice to be able to mark all of the channels as ready at once, then wait for replies for all of the readers, but this can once again result in deadlocks (if several of the readers are read-blocked, they will send their messages as soon as the channel is marked ready, and flood the C-ring, preventing the sender from activating the rest of them).

Only once this entire process is complete can the writer broadcast the actual data. It is at this point that the efficiency gains over \texttt{PI\_Write} are realized; with all of the readers

\textsuperscript{1}Astute readers will notice a race condition here: a PU may be interrupted by one sender and then receive the ready-message from another before it receives the message from the sender that interrupted it. In practice this is a not a problem; one channel will get marked ready, and shortly after the PU leaves the interrupt handler, it will receive the second interrupt and then the original ready-message, resulting in both channels being correctly marked.
waiting for a message with the same tag — the tag of the bundle itself — the sender can
send a single broadcast message, which will then be received by all of the readers.

The other Pilot collective operations, PI_Gather, PI_Scatter, and PI_Reduce, are
notably missing from this section. Due to time constraints, they were not implemented in
this AutoPilot prototype. Section 6.2 discusses possible implementations for these opera-
tions in a future version.

4.5 Error Handling

The implementation of error handling is straightforward. PI_Abort uses printf to
emit an appropriate error message (which will be displayed by the debugger, if one is
attached to the board or if the program is running in the simulator), and then halts the chip.
If it is called from the CP, it calls ix_halt directly; if called from a PU — which cannot
halt the entire chip — it instead sends an interrupt message to the CP, triggering an interrupt
handler that halts the chip instead. This does lead to a race condition where, if multiple PUs
all call PI_Abort at once, it is possible for the messages from more than one of them to
appear before the first interrupt reaches the CP; however, this is harmless.

Internally, AutoPilot uses the PI_ASSERT macro for error checking. The expansion
of this macro depends on the value of the PI_DEBUG preprocessor definition; if it is 0,
PI_ASSERT is actually omitted entirely from the generated code. If it is 1 or more, however,
PI_ASSERT compiles into a check against a condition which, if it fails, calls PI_Abort,
passing it file and line number and an error message.

When PI_DEBUG is 2 or more, tracing of AutoPilot call sites is enabled. This is accom-
plished by making the public version of each PI_ function actually a macro. If PI_DEBUG is less than 2, this macro simply expands to a call to the real function. If 2 or more, however, the macro prefixes the function call with a call to PI_TRACE, passing it the file and line number. PI_TRACE records both in a pair of arrays; each PU (and the CP) addresses a different entry in each array, ensuring that each process can trace calls into AutoPilot without conflicts.

Printing this information is handled by PI_Abort; when tracing is enabled, additional code is compiled into PI_Abort to report the trace information for the calling process before halting the chip.

4.6 Upper Memory DMEM Allocation

The current version of AutoPilot can only create process DMEM in the lowest 16MB page of memory. This inherently limits the size of per-process DMEM to 512KB in the case of 32 PUs (the greatest number obtained from the maximal 128-core chip). In practice, it will be less, as that page is also shared with program code and CP data memory. In principle, a programmer might like to allocate process DMEM from so-called upper memory, which can be as large as 256MB (divided into 16MB pages).

AutoPilot does not presently support this; it was ultimately deemed to be not worth the effort of implementation for the first, prototype version of AutoPilot. Earlier designs, however, considered support for it, and would have had to make implementation changes to support it. Section 6.4 discusses mechanisms for supporting this in the future; this chapter discusses the original implementation that was considered to support this.
The fundamental issue that must be addressed is that, when PU DMEM is allocated in an upper-memory page, it cannot directly access any memory outside that page, including anything in CP DMEM - such as channels and bundles created by the CP during the configuration phase. This means that some sort of PU-local storage not tied to CP DMEM would be necessary, and channel and bundle information would need to be copied to it.

lib1dc provides two functions for managing such storage, *ix_malloc* for allocating memory from DMEM in a normal malloc-style fashion, and *ix_pu_local_area* for allocating memory “under” the stack. Both of these restrict themselves to allocating memory from DMEM, and are thus safe to use in circumstances where the validity of anything outside DMEM cannot be guaranteed. However, both also return a pointer to the allocated space, which still needs to be stored somewhere.

The approach considered for use was to use *ix_pu_local_area* normally to allocate space during PU initialization, and store a pointer to that space at a known location on the stack. This is a similar approach to what is used by lib1dc for per-PU malloc information and other PU-specific storage: the *Ix_start*pu functions push this information onto the PU stack before the PU even begins executing, and code running on the PU can later recover it by reading the *dstlkm* register, which determines the position of the bottom of the stack, and applying an offset to the result.

The only remaining problem in this implementation would be mapping (global) PI CHANNEL and PI BUNDLE values to entries in this PU-local storage; this would be accomplished by making all PI_* functions macros that actually took the address of the channel and bundle variables, allowing them to look them up by address rather than needing to access the underlying storage. This was ultimately rejected as being unnecessarily cumber-
some in support of a feature that it is not immediately necessary to support; if at some point support for upper-memory DMEM allocation does need to be added, Section 6.4 discusses a more elegant approach suggested by S. Kyo et al.
Chapter 5

Results

This chapter analyzes the AutoPilot library prototype, comparing source code size, object code size, memory usage, and execution time for various AutoPilot operations compared to their equivalents using lib1dc directly. Section 5.1 examines the cost in object code size and memory usage of using AutoPilot, and Section 5.2 examines the performance overhead of using AutoPilot for assorted basic operations. Finally, Section 5.3 compares several implementations of the classic N-Queens implementation, with and without AutoPilot, in terms of performance and code size.

It is important to remember that this is a prototype implementation. The focus of development was on correctness first and foremost, and no effort has been made to optimize performance. Once AutoPilot’s usefulness as an API is shown, there is plenty of room for performance improvements.

5.1 Code Size

The IMAPCAR2 has several megabytes of programmable read-only memory for program code, which is accessible by both the CP and the PUs; consequently, minimizing program size does not carry the same great importance that it does on platforms like the
Cell [24]. Nonetheless, it is desirable to minimize program size; the smaller the program, the less time it takes to load onto the IMAPCAR2, the more programs it can store in EEPROM simultaneously, and the less pressure there is on the program cache on both the CP and the PUs.

Unfortunately, evaluating library size accurately is difficult. The traditional means is the program size, which reports code and data sizes for object files and libraries. However, the standard Linux size program does not support the object file format used by the IMAPCAR2 SDK, and the SDK does not include a version of size that does. The size of the library on disk — and the object files that comprise it, as the IMAPCAR2 SDK uses the standard ar format, in which static libraries are simple archives of object files – can be examined, but this is not an accurate indicator of loaded program size, as the files may take up much more space on disk than they do in memory (e.g., due to debugging information that gets stripped before loading), or much less (e.g., due to large regions of uninitialized data).

The IMAPCAR2 debugger and simulator have the commands prog and map, which will display information about code and data sizes for loaded programs (in different formats, but reporting the same information); however, these display information for the entire program rather than individual components of it, meaning that they give a total size that includes application code, memory used by AutoPilot, and memory used by other libraries, with no indication of how much size is used by each component. Furthermore, when linking against static libraries, the linker will (by default) entirely omit object files that are not referenced by application code or other libraries, so there is no guarantee that such a count includes all of the library footprint.
The most straightforward way of determining library size, then, is to examine the .info files emitted by the compiler as debugging aids. The format for these files is undocumented, but mostly self-explanatory, and they end with a summary of memory usage of the corresponding object file. This is not ideal, however; in particular, .info files are not available for the standard libraries that come with the development kit, making it impossible to compare the size of AutoPilot with the size of other libraries such as pthreads.

The next best approach is the use of progl or map in the debugger. The limitations of this approach mentioned earlier can be worked around; in particular, the linker can be instructed to link all object files and their contents, rather than just those needed for the program. By comparing the size of a program linked only with lib1dc to the size of identical programs linked against pthreads and AutoPilot, the size overhead of AutoPilot can be determined accurately.

Table 5.1 shows size information — whole-program sizes for programs linked against libpthreads, libautopilot, and neither, and calculated library sizes for libpthreads and libautopilot. All sizes are in bytes; text is the size of program code, data is the size of initialized data in main memory, and dbss is the size of uninitialized data in main memory. (Information on initialized and uninitialized data in PE IMEM is also reported by the debugger, but is omitted from this table, as none of these libraries make use of it.) Total DMEM is the total amount of data memory (both initialized and uninitialized) used by the library.

It is important to emphasize that this is worst-case size. In practice, the linker will omit object files not used by any other file in the link; this means that only those files containing code that is actually called somewhere will be linked in. For example, a program that does
not use any collective operations will not link bundle.io and will thus be approximately 4K smaller, in total, than a program that does. Table 5.2 shows a list of all object files that make up the AutoPilot library, and what public (i.e., callable by application code) functions each contains.

That said, it’s clear that neither pthreads nor AutoPilot add much code to a program, compared to the size of the standard library. pthreads adds 12%; AutoPilot, 8%. DMEM is a different story; AutoPilot uses nearly 9K of DMEM for bookkeeping structures, increasing the library memory footprint (relative to lib1dc alone) by 43%. This pales compared to pthreads, however, which requires 40K of DMEM and thus nearly triples the size of the program compared to using only lib1dc. It can be stated that, from the standpoint of memory usage, AutoPilot is quite an economical library.

<table>
<thead>
<tr>
<th>Libraries</th>
<th>text</th>
<th>ddata</th>
<th>dbss</th>
<th>Total DMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1dc only</td>
<td>134944</td>
<td>14748</td>
<td>5824</td>
<td>20572</td>
</tr>
<tr>
<td>1dc+pthreads</td>
<td>151344</td>
<td>17812</td>
<td>42788</td>
<td>60600</td>
</tr>
<tr>
<td>1dc+autopilot</td>
<td>145558</td>
<td>21504</td>
<td>7812</td>
<td>29316</td>
</tr>
<tr>
<td>pthreads only</td>
<td>16400</td>
<td>3064</td>
<td>36964</td>
<td>40028</td>
</tr>
<tr>
<td>autopilot only</td>
<td>10614</td>
<td>6756</td>
<td>1988</td>
<td>8744</td>
</tr>
</tbody>
</table>

Table 5.1: Library size comparison
<table>
<thead>
<tr>
<th>File</th>
<th>Purpose</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort.io</td>
<td>Error Handling</td>
<td>PI_Abort</td>
</tr>
<tr>
<td>bundle.io</td>
<td>Collective Operations</td>
<td>PI_CreateBundle; PI_Select; PI_TrySelect; PI_Broadcast</td>
</tr>
<tr>
<td>channel.io</td>
<td>Channel Operations</td>
<td>PI_CreateChannel; PI_CopyChannels; PI_ChannelHasData; PI_Read; PI_Write</td>
</tr>
<tr>
<td>configure.io</td>
<td>Configuration Phase</td>
<td>PI_Configure; PI_CreateProcess</td>
</tr>
<tr>
<td>format.io</td>
<td>Format String Parsing</td>
<td>none</td>
</tr>
<tr>
<td>readwrite.io</td>
<td>C-Ring Messaging</td>
<td>none</td>
</tr>
<tr>
<td>startstop.io</td>
<td>Execution Phase</td>
<td>PI_StartAll; PI_StopMain; PI_StartTime; PI_EndTime</td>
</tr>
</tbody>
</table>

Table 5.2: Object File Contents
5.2 Performance Impact

This section discusses the performance impact of AutoPilot on basic operations — PU activation and message passing — relative to using the lib1dc API directly. These are not whole-program tests, but rather, microbenchmarks designed to compare individual actions. The source code is also compared, in an attempt to compare the ease of use of the two APIs. Tests were run to assess the performance of mode switching (PI_Configure, PI_StartAll and PI_StopMain), simple messages (PI_Write with a single int as payload), broadcast messages (PI_Broadcast), and “large” messages (PI_Write with an array of ints as payload).

All tests are run with a “fast” build configuration: full compiler optimization, no debug symbols, and PI_DEBUG=0 — i.e., no internal error checks, and no logging or call tracing. Non-AutoPilot code refrains from checking the return values of Ix_start_pu and related functions, which is the closest equivalent. (Leaving error checks in would increase the gap in source code size between implementations, as AutoPilot automatically handles error checking that lib1dc-based programs must perform by hand, but would also interfere with performance evaluation.)

Tests were run the IMAPCAR2 simulator in “run” mode with accurate memory timings enabled. This greatly simplifies the process of running the tests, as it does not require a physically connected test board (or drivers for said board), and gives results equivalent to running the tests on the actual hardware (assuming the default maximum clock speed of 133MHz). The drawback is that it is on the order of 4000 times slower than running the same code on the physical device (using a 2.4GHz Intel Core 2 processor); however, the
gains in test automation and result collection far outweigh the time lost due to the slower simulation speed.

5.2.1 Mode Switching

These tests compare the cost of switching from SIMD to MP or mixed mode and back again, for varying numbers of PUs. This means starting the required number of PUs, assigning each a dummy function to execute (which will return immediately), waiting for all of the PUs to finish execution, and then returning the system to SIMD mode. Three approaches are compared: using AutoPilot, using \texttt{Ix\_start\_pu} in a loop, and using a single call to \texttt{Ix\_start\_multi\_pu}. When using lib1dc, the dummy function is passed to the PU directly. When using AutoPilot, \texttt{PI\_CreateProcess} is called an appropriate number of times after \texttt{PI\_Configure} to assign the function to all active PUs. The results are shown in Figure 5.1; there is no time recorded for \texttt{Ix\_start\_multi\_pu} with \texttt{PUNO = 32} because (as noted earlier) \texttt{Ix\_start\_multi\_pu} does not support starting all 32 PUs at once.

It is clear from these results that the performance impact of AutoPilot is significant, especially as the number of PUs increases. With small PU counts, the performance hit is a negligible $5\mu s$; by the time it reaches 32 PUs, however, AutoPilot is imposing a cost of $99\mu s$ relative to a simple \texttt{Ix\_start\_pu} loop, a 54\% increase. Compared to \texttt{Ix\_start\_multi\_pu}, it’s even worse — $112\mu s$, or 68\%. This performance penalty can be attributed to the fact that the Pilot approach creates the \texttt{PI\_PROCESSES} separately, and requires a C-ring message for each process created (to assign the work function and its arguments to a PU). The lib1dc implementations assign the work functions at the same time.
they initialize the PUs, without passing C-ring messages (and without passing arguments to the work functions).

This performance hit, however, is negligible compared to the cost of processing an entire frame — in the worst case, AutoPilot’s overhead amounts to just over 0.3% of the total time \(33\frac{1}{2}\)ms alloted for each frame. It is significant when compared to \texttt{Ix\_start\_pu} and \texttt{Ix\_start\_multi\_pu}, but all three are still so fast as to be inconsequential in typical image processing usage, where time limits are measured in tens of milliseconds.

### 5.2.2 Simple Messages

These tests compare the performance of message-passing — specifically, “simple” messages, consisting of an individual int sent one-to-one (although any other supported data type could be used equally well). The performance of sending arrays (using formats
like `%16d` or of broadcasting is examined later.

For this test, $N$ PUs are started, and a single int is sent to each one — using loops around `PI_Write` and `PI_Read` for the AutoPilot version, and `ix_send_msg` and `ix_recv_msg` for the lib1dc version. The benchmark application then waits for the PUs to exit using `Ix_join` before recording the total execution time and, subsequently, returning the system to SIMD mode. The results are presented in Figure 5.2. Time shown is total time, i.e., the total amount of time required to send one message to each active PU. Experimentation shows that the results are not dependent on the order in which PUs are activated or communicated with.

AutoPilot is significantly slower than lib1dc, and as the number of active PUs increases, so does the time it takes to send each message. Profiling — the results of which are summarized in Table 5.3 — shows that the sender spends most of its time waiting for
<table>
<thead>
<tr>
<th>Operation</th>
<th>% Time (Write)</th>
<th>% Time (Read)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read channel information</td>
<td>2%</td>
<td>1%</td>
</tr>
<tr>
<td>Parse format string</td>
<td>8%</td>
<td>4%</td>
</tr>
<tr>
<td>Check bundle membership</td>
<td>8%</td>
<td>3%</td>
</tr>
<tr>
<td>RTS interrupt &amp; message</td>
<td>3%</td>
<td>N/A</td>
</tr>
<tr>
<td>Wait for handshake</td>
<td>72%</td>
<td>56%</td>
</tr>
<tr>
<td>Write back format info</td>
<td>N/A</td>
<td>16%</td>
</tr>
<tr>
<td>Check format consistency</td>
<td>2%</td>
<td>N/A</td>
</tr>
<tr>
<td>Send/receive data</td>
<td>5%</td>
<td>20%</td>
</tr>
</tbody>
</table>

Table 5.3: Profiling results for simple message passing

the receiver to reply to the RTS message. The receiver, meanwhile, spends some time waiting for the RTS message itself, but takes a long time to reply due to the cache writeback required to share the results of its format string parsing with the sender — it spends most of its time waiting for memory. Furthermore, as the number of activated PUs is increased, the amount of time required to send each message also increases. The profiler shows that PUs with higher IDs take longer to perform the cache writeback, probably — although this has not been verified — due to being positioned further away from the memory controller on the M-ring. The lib1dc implementation, as it does not depend on external memory access, is unaffected by this.

The profiling results are based on a 32-PU performance test, using the combined time for each operation across all messages sent and received; PI_Read is called before
PI_Write, so the reader does not find the channel immediately ready. Items listed as “N/A” do not apply to that side of the communication.

*Read Channel Information* is the initial unpacking of the channel value, *parse format string* the parsing of the format string and construction of an array containing the results of the parse, and *check bundle membership* a request to external memory to see if the channel being used is part of a bundle; these all happen before any messages are sent or received. *RTS interrupt & message* is the sending of the interrupt and information message that will mark the channel ready on the reader’s side; at this point, the writer is *waiting for handshake* until the reader can reply with an RTR message. The reader, meanwhile, has been waiting in the same state since it finished the bundle membership check; as soon as the interrupt is received, it initiates the *cache writeback of the format string parse results* and then sends the RTR message back to the writer. At this point, the reader is once again waiting, this time to *receive data*; the writer, as soon as it receives the RTR message, does a consistency check to ensure that the format strings match and, if they do, sends one or more messages containing the actual data before returning.

It is evident that the vast majority of the time is spent simply waiting for messages to arrive; after that, the greatest performance penalty comes from the cache writeback that the reader must perform (while the writer waits for it) to ensure that the results of parsing the format string are in main memory, so that the writer can compare them to its own format string and ensure that both formats match.
5.2.3 Broadcasts

These tests compare the performance of broadcasting a single int to a variable number of PUs. The approach taken here is analogous to that used for timing PI_Write — N PUs are activated, the message is broadcast to all of them, the PUs exit as soon as they receive the messages, and the CP waits for them to exit using Ix_join. The results are presented in Figure 5.3.

The results here are expected. Broadcasting, like PI_Write, is comparatively slow, taking roughly a millisecond to broadcast to all of the PUs. This is faster than calling PI_Write on each channel individually, but only just; while the actual data transmission can take advantage of the IMAPCAR2’s hardware support for broadcasts, it must interrupt and perform the deadlock avoidance handshake with each PU individually first, a process that receives no benefit from hardware broadcast support. Consequently, the improvement
over individual PI_Write calls is small, and, like PI_Write, as the number of active cores increases, the per-core cost to send a message increases as well.

5.2.4 Large Buffers

These tests compare the performance of sending large buffers, i.e., messages too large to fit in a single C-ring message. In Pilot terms, this means messages containing arrays, with format strings like "%512u". In lib1dc terms, this is any message larger than three uint32s, sent using ix_send_msg_blk or ix_send_msg_blk121, where “121” stands for “one-to-one”. (The distinction between the two is that that ix_send_msg_blk uses broadcast messages, while the -121 variant does not; this means that ix_send_msg_blk121 cannot be used to implement scatter-type operations, but can be used to send messages to the CP, which cannot receive broadcasts.)

AutoPilot sends these messages by copying them into EMEM and then sending a pointer via the C-ring, which the receiver then copies into the destination buffer; lib1dc by packing the entire buffer into a sequence of eight-byte messages, along with sequence information so that they can be correctly received even if they arrive out of order, and sending them all via the C-ring.

The results are presented in Figure 5.4, based on sending a buffer ranging from 4B to 2KB to PU 0. 2KB was chosen as the upper bound on buffer size as 4KB, being the size of the entire data cache, would fill the cache, potentially causing cache misses and interfering with timing. Unsurprisingly, it gets slower as the buffer size increases. The slowdown is linearly proportionate to the buffer size. It turns out that, as the size increases, increases in buffer size cost more for the AutoPilot version than for the lib1dc version,
Figure 5.4: Performance: Buffers

independently of the constant overhead of format string parsing and C-ring flood avoidance. It is likely that AutoPilot could get improved performance by switching from EMEM copies to ix_send_msg_blk121 for large messages.

5.2.5 Source Code

The complete source code listing for the performance tests in this chapter is included in Appendix C. In general, there is not a significant difference between source code sizes; the AutoPilot version is slightly more concise. The primary difference is in the clarity of the code.

Figure 5.5 is a side-by-side comparison of the code used to enter and leave MP mode in both AutoPilot and lib1dc. AutoPilot requires more per-process setup, with the separate calls to PI_CreateProcess and PI_CreateChannel; however, it also automatically
handles the division of the provided DMEM area into individual DMEM regions (and, if a 
DMEM area were not provided, would automatically allocate one). Furthermore, it auto-
matically chooses between mixed mode and MP mode based on the number of processes 
requested.

Figure 5.6 is a comparison of the code used to send and receive messages — the for-
mer, the loop used by the CP to send a message to each process, and the latter, the code 
that executes on each PU to receive the message from the CP. Here, the difference in size 
is more evident. The AutoPilot version, additionally, frees the user from needing to sepa-
rately allocate and configure an ix_MSG structure for each message send and receive, and 
— less obviously — automatically performs type checking, deadlock avoidance, and chan-
nel endpoint checking to guarantee that the sender and receiver agree on the type of the 
message, and that it is delivered to the correct process without deadlocking the program.
As is customary with Pilot, the `PI_Write` and `PI_Read` syntax can be instantly grasped by analogy with C’s `stdio` functions, whereas the `lib1dc` programmer will need to consult documentation concerning three functions’ arguments and how to correctly fill in the message header.

### 5.3 Case Study: Parallel N-Queens

The IMAPCAR2 SDK comes with a number of code examples, demonstrating the usage of the chip in both SIMD and MIMD modes. One of these examples is an implementation of the *N-Queens Problem*. This is a classic puzzle in which one is tasked to arrange \( N \) chess queens on an \( N \times N \) board such that none of the queens are capable of attacking each other — i.e., none of them share a row, column, or diagonal with any other. This is also known as the *eight queens puzzle*, after the most common incarnation of it (which uses
a standard $8 \times 8$ chessboard, and thus, eight queens).

There are three implementations of the N-queens solver that come with the SDK: a sequential version, which performs all computation on the CP; a SIMD version, which uses the PE array as much as possible; and a MIMD version, which uses the PUs and the message passing API. All of these use a straightforward recursive brute-force approach, by placing a queen in each possible location on the board and, for each location, recursing to solve the case for $n - 1$ queens with one queen already on the board, and so forth, with the base case consisting of finding every place on the board where one can put the final queen with the other seven already placed. The parallel version works by assigning each one-queen position to a PU to calculate the potential solutions for; as there are more possible positions for the first queen than there are PUs, PUs are assigned new positions to solve as they finish the old ones.

AutoPilot adds two implementations to this, both based on the MIMD version and using the AutoPilot API. One uses simple round-robin scheme for distributing work orders to the PUs, and the other assigns work to each PU as it becomes ready, regardless of which order it happens in, in a manner similar to that of the MIMD version. The MIMD version uses a slightly more efficient work dispatch feature in which each work order is sent in a message that will be automatically received by the first available PU (in contrast to the Pilot non-round-robin version, which must wait for an explicit message from each PU indicating that it is ready); this is a feature not yet exposed by AutoPilot, but proposed as an extension to PI_Broadcast in Section 6.2. This difference in work order allocation schemes is the only difference — apart from choice of message-passing API — between the MIMD and AutoPilot implementations. The algorithm used to actually solve the problem, and the
division of that problem into work orders for the PUs, is identical, as is the choice of message-passing over shared memory.

It is worth noting that the MIMD implementation included in the SDK was found to contain bugs. In particular, when run with $PENO = 16$, the PUs were not shut down properly, and thus never wrote their results back to external memory, causing the program to report incorrect results. Furthermore, due to edge cases when configured to use all 32 PUs, it was observed to deadlock. Both of these problems have been fixed in the version tested here. This does, however, underscore the ease with which even experienced IMAPCAR2 programmers can fall prey to the pitfalls of the lib1dc API.

Despite the fact that the IMAPCAR2 is primarily an image processing chip, no image processing benchmarks were tested. The reason for this is twofold. The first is the lack of readily available MIMD image processing benchmarks for the IMAPCAR2. No such programs were included in the SDK, and the source code used for the benchmarks discussed by the developers in their publications is not available, nor is the code for proprietary real-world applications using this chip. Thus, any image processing benchmarks would have to be written from scratch — by someone with enough image processing knowledge to verify that they were correctly implemented.

The second reason is that AutoPilot is, by design, not concerned with the image processing specific features of the chip in any case. It is meant to replace the process control and communication features of the standard library, leaving the rest of a program — whether doing region of interest analysis or chess piece placement — untouched. Given this, a benchmark like N-Queens that puts the processor under heavy load without any possibility of bottlenecking on external resources, such as the video decoder or EMEM image
buffers, is ideal, making it easy to assess how much overhead AutoPilot adds, and how badly that affects performance under varying ratios of communication to computation.

5.3.1 Code Size

The relative sizes of the different implementations are presented in Table 5.4. The methodology used here is the same as in Table 5.1; the programs are loaded into the simulator and examined with the prolog command. As with the prior table, sizes given are whole-program size, application code and library code combined. The increase from using AutoPilot is less than the library size measured earlier because not all components of AutoPilot are used by application code, and thus, not all are included in the final program image by the linker.

AutoPilot increases the size of the programs. Proportionally, the size increase is much greater than the results of the library size comparison would imply; this is because linker optimization permits most of the standard library to be omitted. Comparatively less of

<table>
<thead>
<tr>
<th>Implementation</th>
<th>text</th>
<th>ddata</th>
<th>dbs</th>
<th>Total DMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>6928</td>
<td>622</td>
<td>512</td>
<td>8062</td>
</tr>
<tr>
<td>SIMD</td>
<td>7114</td>
<td>622</td>
<td>0</td>
<td>7736</td>
</tr>
<tr>
<td>MIMD</td>
<td>9120</td>
<td>2058</td>
<td>512</td>
<td>11690</td>
</tr>
<tr>
<td>AutoPilot (RR)</td>
<td>14589</td>
<td>4196</td>
<td>1156</td>
<td>19941</td>
</tr>
<tr>
<td>AutoPilot (non-RR)</td>
<td>16334</td>
<td>4708</td>
<td>2180</td>
<td>23222</td>
</tr>
</tbody>
</table>

Table 5.4: N-Queens Size Comparison
AutoPilot can be omitted, and linking it also causes other parts of the standard library that it depends on to be linked in. The non-round-robin AutoPilot version is larger than the round-robin version because it uses collective operations, causing a few additional object files to be linked in.

5.3.2 Performance

8 Queens

The first performance test used the classic 8 Queens Puzzle. Testing methodology was the same as Section 5.2 — the programs were built with \texttt{PI\_DEBUG=0} and all logging and error checking disabled, and run in the simulator. Times were recorded with \texttt{IxPtime}.

The sequential version, of course, does not vary in runtime as the number of PUs changes, as it does not use them; it finishes in 2471 $\mu$s. The SIMD version shows some variation as more PEs are made available, but the developers note that N-Queens is not a problem that lends itself well to a SIMD approach, and the results bear this out; the best-case performance is 2029 $\mu$s, a speedup of just 1.22. The sequential version’s time is used as the basis time for calculating the speedup of the parallel versions.

Figure 5.7 shows the results. The MIMD version does quite well, peaking with a speedup of just below 10 (at 22 PUs). Beyond that point, there are more PUs than it can reasonably keep busy, and thus, adding more PUs actually makes things worse — the additional PUs don’t actually confer any benefit, but do add communication overhead.

The AutoPilot version, unfortunately, is not just worse than the MIMD version, it’s worse, in many cases, than the sequential version. Communication overhead is so great,
compared to the cost of the actual computations being performed, that any performance gains from running multiple threads are almost completely wiped out by the cost of communicating with the PUs in the first place. Table 5.5 shows the results in more detail, with the best times highlighted. AutoPilot is at its best at 4 PUs, with a speedup of 1.4; as more PUs are added, performance degrades, becoming worse than the sequential version once 19 or more PUs are activated. The difference between the RR and non-RR versions is small but noticeable, with the RR version performing worse in every case and achieving a best-case speedup of 1.28.

10 Queens

The fundamental problem here is that the ratio of useful computation to communication is too low for AutoPilot to be of any help, due to the high fixed cost of communication. In that case, then, we should see an improvement if the amount of computation to be per-
<table>
<thead>
<tr>
<th>PUs</th>
<th>MIMD</th>
<th>AutoPilot_RR</th>
<th>AutoPilot</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1292.92</td>
<td>2466.564</td>
<td>2432.602</td>
</tr>
<tr>
<td>3</td>
<td>891.39</td>
<td>1948.759</td>
<td>1863.421</td>
</tr>
<tr>
<td>4</td>
<td>684.59</td>
<td><strong>1931.286</strong></td>
<td><strong>1771.308</strong></td>
</tr>
<tr>
<td>5</td>
<td>562.42</td>
<td>1964.474</td>
<td>1805.842</td>
</tr>
<tr>
<td>6</td>
<td>496.33</td>
<td>1988.925</td>
<td>1855.85</td>
</tr>
<tr>
<td>7</td>
<td>434.92</td>
<td>2054.195</td>
<td>1892.128</td>
</tr>
<tr>
<td>8</td>
<td>395.64</td>
<td>2119.361</td>
<td>1939.489</td>
</tr>
<tr>
<td>9</td>
<td>359.67</td>
<td>2142.752</td>
<td>1985.383</td>
</tr>
<tr>
<td>10</td>
<td>335.8</td>
<td>2213.602</td>
<td>2031.233</td>
</tr>
<tr>
<td>11</td>
<td>316.19</td>
<td>2242.902</td>
<td>2089.038</td>
</tr>
<tr>
<td>12</td>
<td>308.48</td>
<td>2295.128</td>
<td>2132.278</td>
</tr>
<tr>
<td>13</td>
<td>300.24</td>
<td>2352.481</td>
<td>2188.699</td>
</tr>
<tr>
<td>14</td>
<td>284.57</td>
<td>2396.97</td>
<td>2235.504</td>
</tr>
<tr>
<td>15</td>
<td>271.75</td>
<td>2462.94</td>
<td>2290.459</td>
</tr>
<tr>
<td>16</td>
<td>263.48</td>
<td>2501.857</td>
<td>2324.992</td>
</tr>
<tr>
<td>17</td>
<td>257.17</td>
<td>2577.962</td>
<td>2329.737</td>
</tr>
<tr>
<td>18</td>
<td>255.2</td>
<td>2623.707</td>
<td>2387.774</td>
</tr>
<tr>
<td>19</td>
<td>256.55</td>
<td>2709.173</td>
<td>2476.18</td>
</tr>
<tr>
<td>20</td>
<td>253.57</td>
<td>2766.256</td>
<td>2510.383</td>
</tr>
<tr>
<td>21</td>
<td>251.44</td>
<td>2817.338</td>
<td>2562.338</td>
</tr>
<tr>
<td>22</td>
<td><strong>249.37</strong></td>
<td>2873.271</td>
<td>2599.571</td>
</tr>
<tr>
<td>23</td>
<td>251.21</td>
<td>2935.376</td>
<td>2628.549</td>
</tr>
<tr>
<td>24</td>
<td>253.79</td>
<td>2997.654</td>
<td>2682.571</td>
</tr>
<tr>
<td>25</td>
<td>256.01</td>
<td>3050.692</td>
<td>2804.729</td>
</tr>
<tr>
<td>26</td>
<td>259.12</td>
<td>3105.248</td>
<td>2831.632</td>
</tr>
<tr>
<td>27</td>
<td>260.83</td>
<td>3173.774</td>
<td>2914.436</td>
</tr>
<tr>
<td>28</td>
<td>260.25</td>
<td>3231.105</td>
<td>2920.564</td>
</tr>
<tr>
<td>29</td>
<td>255.58</td>
<td>3278.338</td>
<td>2978.248</td>
</tr>
<tr>
<td>30</td>
<td>258.32</td>
<td>3338.774</td>
<td>3066.797</td>
</tr>
<tr>
<td>31</td>
<td>258.03</td>
<td>3378.323</td>
<td>3021.18</td>
</tr>
<tr>
<td>32</td>
<td>262.23</td>
<td>3424.744</td>
<td>3101.481</td>
</tr>
</tbody>
</table>

Table 5.5: Runtime details for 8 Queens
formed is increased, provided that the increase in required communications is not as severe — thus skewing the ratio more towards computation. Fortunately, this is easy to do.

The second performance test solved the version with $N = 10$ — ten queens on a $10 \times 10$ board. This dramatically increases the amount of computation each thread needs to perform before it can request a new work assignment. The number of work orders increases from 42 to 72, not quite doubling; however, the total amount of computation required increases by a factor of 20, with the serial version taking 50ms to execute. (The number of possible board configurations increases by a factor of 3,910; the algorithm used, however, does not examine every possible configuration.)

The results are shown in Figure 5.8 and Table 5.6. The MIMD version is still faster, unsurprisingly, with a best-case speedup of 24. As predicted, however, AutoPilot does much better as well. The peak speedup of AutoPilot is now 12.4, with 24 PUs active, and there is more differentiation between the RR and non-RR versions — although which one comes out on top depends on whether it can keep all of the PUs saturated or not, which means that as the non-RR version starts to get more PUs than it can keep active, the RR version pulls ahead.

### 5.3.3 Source Code

The source code for the N-Queens implementations used in this chapter is available in Appendix D. As with the benchmarks, the differences in code size are not significant, with the bulk of the source code consisting of computation unaffected by choice of library. The difference, yet again, is that the AutoPilot code is more immediately readable than the lib1dc version, and that it performs much more safety checking “under the hood” that,
<table>
<thead>
<tr>
<th>PUs</th>
<th>MIMD</th>
<th>AutoPilot_RR</th>
<th>AutoPilot</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>25074.06</td>
<td>26988.218</td>
<td>26915.474</td>
</tr>
<tr>
<td>3</td>
<td>17018.38</td>
<td>18540.707</td>
<td>18423.632</td>
</tr>
<tr>
<td>4</td>
<td>12755.81</td>
<td>14358.135</td>
<td>13953.293</td>
</tr>
<tr>
<td>5</td>
<td>10404.23</td>
<td>11903.398</td>
<td>11389.895</td>
</tr>
<tr>
<td>6</td>
<td>8541.39</td>
<td>9664.241</td>
<td>9493.226</td>
</tr>
<tr>
<td>7</td>
<td>7393.54</td>
<td>8728.707</td>
<td>8345.744</td>
</tr>
<tr>
<td>8</td>
<td>6565.11</td>
<td>7878.805</td>
<td>7523.038</td>
</tr>
<tr>
<td>9</td>
<td>5860.97</td>
<td>7069.414</td>
<td>7057.842</td>
</tr>
<tr>
<td>10</td>
<td>5434.98</td>
<td>7019.429</td>
<td>6668.353</td>
</tr>
<tr>
<td>11</td>
<td>4924.13</td>
<td>6226.135</td>
<td>6026.564</td>
</tr>
<tr>
<td>12</td>
<td>4467.62</td>
<td>5550.06</td>
<td>5500.323</td>
</tr>
<tr>
<td>13</td>
<td>4261.2</td>
<td>5537.699</td>
<td>5327.805</td>
</tr>
<tr>
<td>14</td>
<td>3912.32</td>
<td>5322.436</td>
<td>5088.03</td>
</tr>
<tr>
<td>15</td>
<td>3588.41</td>
<td>4895.827</td>
<td>4766.414</td>
</tr>
<tr>
<td>16</td>
<td>3496.61</td>
<td>4959.248</td>
<td>4673.338</td>
</tr>
<tr>
<td>17</td>
<td>3377.9</td>
<td>4880.752</td>
<td>4739.278</td>
</tr>
<tr>
<td>18</td>
<td>3113.72</td>
<td>4413.421</td>
<td>4609.97</td>
</tr>
<tr>
<td>19</td>
<td>2934.54</td>
<td>4452.534</td>
<td>4324.09</td>
</tr>
<tr>
<td>20</td>
<td>2877.05</td>
<td>4407.376</td>
<td><strong>4226.128</strong></td>
</tr>
<tr>
<td>21</td>
<td>2819.35</td>
<td>4390.323</td>
<td>4301.241</td>
</tr>
<tr>
<td>22</td>
<td>2762.59</td>
<td>4474.361</td>
<td>4312.759</td>
</tr>
<tr>
<td>23</td>
<td>2709.2</td>
<td>4548.917</td>
<td>4312.12</td>
</tr>
<tr>
<td>24</td>
<td>2364.53</td>
<td><strong>4052.556</strong></td>
<td>4309.406</td>
</tr>
<tr>
<td>25</td>
<td>2270.96</td>
<td>4057.451</td>
<td>4352.218</td>
</tr>
<tr>
<td>26</td>
<td>2245.74</td>
<td>4141.368</td>
<td>4449.135</td>
</tr>
<tr>
<td>27</td>
<td>2214.68</td>
<td>4244.15</td>
<td>4496.707</td>
</tr>
<tr>
<td>28</td>
<td>2191.17</td>
<td>4313.414</td>
<td>4514.564</td>
</tr>
<tr>
<td>29</td>
<td>2150.91</td>
<td>4394.346</td>
<td>4636.173</td>
</tr>
<tr>
<td>30</td>
<td>2107.13</td>
<td>4466.481</td>
<td>4323.729</td>
</tr>
<tr>
<td>31</td>
<td><strong>2072.2</strong></td>
<td>4576.827</td>
<td>4358.083</td>
</tr>
<tr>
<td>32</td>
<td>2079.55</td>
<td>4554.789</td>
<td>4471.248</td>
</tr>
</tbody>
</table>

Table 5.6: Runtime details for 10 Queens
in the lib1dc version, must be performed by application code (or omitted entirely, as the example N-Queens implementation that comes with the SDK does).

5.4 Summary

While AutoPilot works, the methods used to avoid C-ring floods, and thus deadlocks, introduced too much overhead, which was particularly detrimental to broadcast efficiency. On the other hand, the size of the library is quite competitive, the source code is clear and readable, and it demonstrates that the AutoPilot API can work on, and is a good fit for, this architecture. While it is not yet ready for real-world use, there is significant scope for performance improvements, which could probably speed it up to the point that it is useful as a production library and not just a proof-of-concept prototype.
Chapter 6

Conclusions and Future Work

AutoPilot was created to provide a simpler, safer API for the IMAPCAR2 and related chips. The Pilot API provides a single, consistent interface for managing process creation and communicating between processes, following printf-style conventions that are immediately familiar to any C programmer; the application programmer no longer needs to worry about choosing between several options for the same operation, or about the edge cases involved in functions such as \texttt{Ix\_start\_multi\_pu}. At the same time, runtime verification of channel endpoints and format strings eliminates the possibility of sending messages to the wrong process or sending the wrong data types, automatic handshaking between processes eliminates the possibility of C-ring floods, and call site tracing makes it much easier to debug errors in application code when they do occur.

In this, it is successful. As seen in Chapter 3, the core features of Pilot are implemented on the IMAPCAR2. The Pilot API is demonstrated to be usable on this architecture with only minor alterations, and as the size comparisons in Chapter 5 show, AutoPilot is also quite compact, making it an easy addition to any program. The debugging features, such as call site tracing, are also effective.

The performance comparison shows that AutoPilot is significantly slower than the libIdc implementation. The overhead of deadlock avoidance, in particular, slows it down to
the point that AutoPilot may be unsuited for real-world applications in its current form; typical performance constraints require each frame to be processed within 33 milliseconds\([42]\), which means that overhead measured in milliseconds rather than microseconds is incapacitating. The exception to this is the actual process of switching to and from MP mode, which is fast enough to be usable and gives AutoPilot some applicability even in cases where its implementation of message passing is too slow to be used. However, with the API itself already proven, performance improvements can be made without a redesign of the library interface.

There are still many improvements that can be made. The most obvious is improvements to message-passing performance, but there is also scope for implementing more collective operations and the deadlock detector from normal Pilot, as well as implementing support for usage of upper memory pages as PU DMEM areas. AutoPilot also imposes some restrictions on the programmer — in particular, they cannot safely use interrupt masking or manual cache invalidation without the risk of interfering with AutoPilot’s C-ring flood avoidance — and these improvements may allow these restrictions to be lifted, too.

The follow sections detail areas of future work. Potential performance improvements in the areas of one-to-one message passing and collective operations are discussed first, as well as improvements that may be enabled by features planned for future revisions of the XC Core architecture. This is followed by a discussion of new features that could be implemented — new collective operations, deadlock detection, and support for locating PE DMEM in upper memory. It concludes with a look at the possibility of implementing the Pilot API on other, similar architectures.
6.1 Performance Improvements

AutoPilot is barely fast enough for basic usage, and can (and should) be faster. This prototype has been constructed primarily with an eye towards ease of implementation and debugging, not performance; for real-world use, a number of improvements can be made to various basic operations.

6.1.1 Message Passing

Message passing with AutoPilot is slow compared to lib1dc. This is attributable primarily to two factors: the cost of the multiple messages sent for the initial synchronization between sender and receiver to avoid C-ring floods, and the cost of the cache writebacks needed to ensure that both processes have a consistent view of the results of format string parsing. If a way can be found to improve either of these, it would dramatically improve Pilot’s performance.

One potential approach for improving synchronization is to do away with the back-and-forth synchronization protocol and use a circulating status message (or messages) that contains information about the other processes. In this, each bit in the message would correspond to a read-ready or write-ready flag for a channel. Processes read-blocked or write-blocked would wait for this message, updating it as necessary with their own information before retransmitting it, and when both ends of the channel were ready, data transmission would occur. The biggest problem with this approach is that messages are limited to 96 bits, and each channel would require two bits (one for read-ready, one for write-ready). This means that two messages would effectively be the minimum practical,
imposing a limit of 96 channels total on the program. It would, however, dramatically reduce the complexity of the C-ring flood avoidance code, and would hopefully improve performance, as well.

Earlier designs for AutoPilot considered ignoring synchronization entirely, and having the sender interrupt the receiver and transmit the entire message, which would be buffered on the receiving size until PT_Read was called. However, the unbounded memory requirements of this approach made it unsuitable.

The actual data transmission could also be improved in some ways. At present it uses a “safe and slow” approach in which each item in the format string corresponds to a single message; communications with many small format string items could pack multiple items per message, and for sending large messages, ix_send_msg_b1k121 is faster than EMEM transfer.

Reducing the amount of external memory access would be the greatest improvement, however. The most obvious way to do this is to reduce the amount of data that needs to be written back, or eliminate the cache writeback entirely. The latter could be accomplished if we assume an additional constraint on format strings passed to the AutoPilot API: they must be constant (i.e., the contents a format string cannot change during the execution of the program, a constraint most easily satisfied by only ever using string literals for format strings). Given this, no results of parsing the format string need to be sent by either party; instead, one of the processes can simply send a pointer to the format string itself to the other, which then parses both format strings and checks the results to see if they are equivalent. This means one process needs to parse the format string twice, but considering how much faster format string parsing is than external memory access, this is still preferable.
Another approach would be to keep the current design, but send the results of the parsing via the C-ring (using \texttt{ix\_send\_msg\_blk121}) rather than via cache writeback. This runs the risk of briefly bottlenecking other processes attempting to send or receive messages but avoids external memory access entirely, and is thus probably still the faster option. It would also be possible to use conditional compilation such that when \texttt{PI\_DEBUG} is 0 (see Section 5.2 and Appendix A), format string verification is dispensed with entirely.

One last (and attractively straightforward) option is simply to replace calls to \texttt{ix\_memcpy\_c2nc} with calls to \texttt{ix\_cwb} or \texttt{ix\_cwba}. The latter two functions are much faster, but also much coarser, with a resolution of one cache line (512 bytes) rather than one byte. This means that, to be safe, any use of them needs to ensure that either overlapping cache writebacks will not occur, or that they will not cause problems. For AutoPilot, in practice, this means ensuring that none of the format string parsing result buffers share a cache line with anything that another process might access and, in particular, that they cannot share a cache line with \textit{each other}. This will probably require adding padding or alignment restrictions (and thus increasing the amount of data memory used) for these buffers.

6.1.2 Collective Operations

Broadcast is, unfortunately, bottlenecked on the requirement to mark each channel as ready individually. None of the setup preceding the actual transmission of data can make use of the IMAPCAR2’s broadcast hardware, and that makes up the bulk of the time spent broadcasting. It is still faster than calling \texttt{PI\_Write} on every channel individually, but only barely.

Doing this efficiently is made problematic by the limitations on interrupt messages
(broadcast interrupts cannot choose which PUs are interrupted). One possible solution
would be to broadcast an interrupt to everyone, using a different “broadcast-aware” inter-
rupt handler, and follow it with a broadcast message containing the bundle information;
PUs that are not part of the affected bundle would simply return, and ones that are would
mark their channel in it ready. This has the (fairly large) disadvantage of briefly interrupt-
ing every PU on the chip, but means setup can be done in \(2 + n\) messages — the two initial
broadcasts, and the \(n\) replies from the receivers — rather than \(3n\).

A less drastic change would be to send out the ready messages in batches. The sender
can send interrupts and ready-messages until the C-ring is, say, half full; it can then wait for
replies, and as it gets each reply, send out another ready message, until all of the channels
are marked ready. This doesn’t reduce the total number of messages required, but it does
mean that the sender won’t end up blocked on a single slow receiver while all the others
are still waiting for their channels to be marked ready, while still avoiding the possibility
of a C-ring flood.

It may be that, like PI_Write itself, PI_Broadcast requires a fundamental rethink of
how deadlock avoidance is implemented.

6.1.3 Fast Interrupts

Correspondence with Shorin Kyo has indicated that future versions of the architecture
will support arguments in interrupt message handlers (as the message currently has a 12-
byte payload which is discarded unused when the message is used to invoke an interrupt).
This would immediately simplify the C-ring flood avoidance code by allowing it to dispense
entirely with the second message that carries channel information, instead providing that
information as arguments to the interrupt handler and eliminating an entire C-ring message from each transmission — or, in the case of bundles, several C-ring messages.

### 6.2 New Collective Operations

There is scope for more, and better, support for collective operations than what AutoPilot presently has. Scatter, in particular, could be implemented with `ix_send_msg_blk` — if the sender can guarantee that all of the receivers are ready before sending the message, it can guarantee that each one receives a different slice of the message (although guaranteeing a consistent mapping between bundle index and slice index may be harder). Performance improvements are likely possible for `PI_Select`, `PI_Broadcast`, and `PI_Gather`, as well.

There is also the possibility of introducing a new collective operation, tentatively called `Distribute`. This takes a message, a write-bundle, and a count, and delivers the message to `count` processes in the bundle. Unlike Broadcast, there is no guarantee that each process gets exactly one message; it may be the case that some processes get a message and some get none, or that some processes even get multiple messages. This would be useful for work-order dispatch, where it doesn’t matter which process in a bundle gets the work order, just that it goes to the first available one — a common structure that currently has to be handled either with a basic round-robin approach, or by listening for messages from the workers signifying that they are ready for more work.

Figure 6.1 shows the proposed API for distribute operations. Rather than implemented as a separate call, it would be implemented as an additional argument to `PI_Broadcast`,
void PI_Broadcast(PI_BUNDLE b,
    size_t count,
    const char * format,
    ...);

// send to first available process
PI_Broadcast(b, 1, "%d", 99);

// send to first 8 available processes
PI_Broadcast(b, 8, "%d", 99);

// normal broadcast
PI_Broadcast(b, PI_BROADCAST_ALL, "%d", 99);

Figure 6.1: Proposed API for distribute operations

dicating how many readers to distribute the message to. The special value PI_BROADCAST_ 
ALL would indicate that it should behave like a normal broadcast operation, i.e., broadcast-
ing to every reader exactly once.

While discussion of Distribute focused mainly on its use for broadcast-like messages,
it would also have been useful for scatter-like messages, and would likely have been added
to PI_Scatter as well. In this usage, it would be passed an array of message values (like 
PI_Scatter) and would send each distinct message once, to be received by the first PU in 
the bundle that is ready for it, with no guarantee that the messages are evenly distributed 
between PUs.

Standard Pilot also implements the reduce operation. Implementing this in AutoPilot
is possible, but implementing it efficiently is a challenge. It is not considered as high a
priority as the other collective operations, as it can easily be implemented in user code,
once PI_Gather is available, by calling PI_Gather and then looping over the results.
6.3 Deadlock Detection

Pilot’s integrated deadlock detector works by starting a separate process that monitors all other communications in the program — whenever a read or a write is performed, a message is sent to the deadlock detector. The detector keeps track of all reads and writes performed and, being aware of the process and channel graph, can detect when a deadlock has occurred (and where) and abort the program. This would not be difficult to implement in AutoPilot, although the performance implications would be significant.

There are two main approaches that could be used. The first is a straightforward implementation of the Pilot approach — when you call `PI_Configure`, it activates one more core than requested, and maps the deadlock detector onto it. When you call any communications function, it first sends a message to the deadlock detector reporting what it is doing. The caller wouldn’t need to do an initial flood-avoidance sync first, since the detector is assumed to always be listening. The deadlock detector then keeps track of the communications graph in the same manner as the Pilot implementation, and calls `PI_Abort` as soon as something goes wrong. This is straightforward and probably has the least performance impact, but requires one core to be entirely dedicated to the deadlock detector.

The alternative approach is to use interrupts. In this design, the CP is responsible for deadlock detection; it enables an interrupt handler that is used by other cores to report communications activity. This is likely to be much slower than the dedicated core approach: not only will the CP have to frequently suspend what it is doing to update the deadlock detector, but (since interrupt messages cannot carry additional arguments) it requires twice
as much message traffic to report each operation — one to trigger the interrupt handler, and one to report the specifics of the operation. The advantage is that it does not require a dedicated core, meaning the communications graph will look exactly as it does in normal operation and thus deadlocks that only occur when specific numbers of cores are active can still be tested for.

6.4 Cross-Page Operation

The IMAPCAR2 can have up to 256MB of memory attached to it. Each PU can only address 16MB of data memory at a time; however, there is no requirement that it be the same 16MB for each PU. The default assumption is that all PUs will use page 0 (physical addresses 0x00000000 through 0x00FFFFFF), and pages above that, if present at all, will be used as image buffers and not accessed directly. In circumstances where it is not possible to fit the data memory regions for all PUs onto a single page, however, it is possible to reserve regions of other pages as PU DMEM and configure some of the PUs to access those pages.

AutoPilot does not currently support this usage. The biggest roadblock is that there is no automatic copying of data between pages, and AutoPilot relies on a number of global structures to track information about channels and processes. Once a PU is addressing a different page, the data stored at those addresses is uninitialized — as is any other global data used by the program, including memory allocation information and string constants! Only the program code remains accessible across pages.

The recommended way of handling this (according to conversations with S. Kyo) is to
copy all (constant and variable) global data to the target page before activating any PUs that use that page, ensuring that any global values the PUs make use of are available at the expected addresses. This is entirely possible, but significantly complicates the configuration phase — not only does it now need to determine the limits of the global data region and copy the data to the target pages, but it also needs modifications to the API to allow user code to specify which page to assign each process to, or to be smart enough to intelligently determine which page each PU should be mapped to automatically.

6.5 Non-IMAP Architectures

While AutoPilot’s code is highly specific to the IMAPCAR2, it shows that the fundamental concepts and API of Pilot are applicable not just to high performance clusters, but also to multicore embedded systems. It is likely that implementations of Pilot for other platforms would be useful. While the IMAPCAR2 is unusual in its runtime reconfiguration features, Pilot’s CSP-based structure would still be a good fit for other multicore architectures such as Visconti [46].

AutoPilot has successfully implemented a safe, simple API on an architecture with a lack of cache coherency and a complex API for interprocess communication. The same approach could be used to bring similar benefits to other architectures with complex IPC mechanisms that would benefit from a simple abstraction layer, and a lack of cache coherency that makes pthreads-style threading APIs unsatisfactory.
Bibliography


Glossary of Terms

1DC Abbreviation for One-Dimensional C (see that glossary entry).

ANSI C The versions of the C programming language standardized by the American National Standards Institute. Also known as “ISO C” (as the standard was later adopted by the International Standards Organization) or “Standard C”.

API Application Programming Interface. The set of publically accessible types and functions used to interface with a piece of software. There can be multiple implementations of the same API; for example, Pilot and AutoPilot.

ASIC Application Specific Integrated Circuit. An integrated circuit implementing a fixed function (in contrast to a general purpose processor, which can be programmed).

C-Ring The ring network connecting the PUs and CP, used for sending messages between processors without needing to use shared memory. There is a hard limit to how many messages can be on the C-ring waiting to be received, equal to half the number of active PUs.

C-Ring Flood A deadlock situation in which the C-ring is full, and all of the processes that would normally be receiving messages from the C-ring are blocked trying to send messages first. This can result in a deadlock even if there is no circular waiting.

Cache Coherency A system has cache coherency if all of the caches provide a consistent (with each other) view of memory. In particular, this requires that memory writes performed by
one processor be reflected on all other processors, either by copying the correspond-
ing cache entry to those processors, or copying it to main memory and invalidating
the corresponding cache entries.

CP  Control Processor. The central RISC processor in an IMAP chip that controls the PEs
and PUs and access to off-chip resources.

CPU  Central Processing Unit. The general purpose processor (or processors) that controls
a computer.

CSP  Communicating Sequential Processes. A formal language developed by C.A.R. Hoare
for describing interactions between processes in concurrent systems.

CTA  Candidate Type Architecture. A model for analyzing parallel computing architec-
tures.

CUDA  A platform for high performance parallel computing using NVIDIA graphics proces-
sors.

DMA  Direct Memory Access. A memory transfer technology that uses a dedicated memory
controller for data transfers between peripherals and main memory, allowing memory
access to occur without interrupting other computation.

DMEM  Data Memory. As CP DMEM, the region of memory usable as stack and heap by the
CP, consisting of all memory below 16MB that is not already taken up by PMEM; as
PU DMEM, the region of memory assigned to a PU as stack and heap space, usually
a small sub-region of CP DMEM.

EEPROM  Electronically Eraseable, Programmable Read Only Memory. Nonvolatile solid-state
storage commonly used to store rarely-changing data such as program code.
EMEM External Memory. Memory above the 16MB line used as scratch space and image buffers; not directly accessible or cacheable by the CP or PUs, but special functions exist to access it.

GPU Graphics Processing Unit. A specialized processor, usually containing hundreds or thousands of SIMD processors connected to high performance RAM, for 2d and 3d graphics rendering. Modern GPUs can also be used for general purpose computing using technologies such as CUDA.

IMAP Integrated Memory Array Processor. A processor architecture for image processing and analysis, consisting of an array of SIMD PEs directly connected to on-chip memory and controlled by an external processor (in early designs) or on-chip CP (in later ones).

IMAP-CE IMAP for Car Electronics. The first single-chip IMAP implementation, marketed for use in “smart cars” and other intelligent vehicles.

IMAP-VISION The first commercial IMAP implementation, consisting of a PCI expansion board containing IMAP components and video encoders and decoders.

IMAPCAR The successor to the IMAP-CE, containing a number of architectural and performance improvements.

IMAPCAR2 The successor to the IMAPCAR, adding MIMD capabilities via dynamic reconfiguration.

IMAP5 Alternate name for the IMAPCAR2, used by some tools in the IMAPCAR2 SDK. The “5” refers to the fact that it is the fifth generation of IMAP-based designs.

IMEM Image Memory. The memory directly attached to the SIMD PEs (as opposed to EMEM, DMEM, and PMEM, which are off-chip).

128
IPC Inter-Process Communication.

ISR Interrupt Service Routine. The function invoked to process an interrupt.

M-Ring A ring network connecting the PUs to the CP’s memory controller. Unlike the C-ring, this cannot be explicitly used by programmers, but is automatically used when a PU requests data from external memory.

MIMD Multiple Instruction, Multiple Data. A form of parallel processing in which each processor can be executing different instructions, on different pieces of data. Contrast SIMD.

MIMD Mode An IMAPCAR2 execution mode in which all PEs have been combined to form PUs, and the chip is functioning as a multi-core processor with no SIMD capabilities.

Mixed Mode An IMAPCAR2 execution mode in which half of the PEs have been combined to form PUs, and the chip is functioning as a multi-core processor with reduced SIMD capabilities available to the CP. (The PUs cannot make use of the SIMD PEs, even in mixed mode.)

MPI Message Passing Interface. A standardized (and de facto standard) API for cluster computing.

N-Ring A ring network connecting the PEs to each other, used in SIMD processing to transfer data between adjacent PEs.

One-Dimensional C A C variant used for IMAPCAR2 programming, consisting of ANSI C with non-standard extensions for SIMD programming.

PCI Peripheral Component Interconnect, a standard for connecting additional components (in the form of internal expansion boards) to a computer.
PE  Processing Element. A single SIMD processor, directly attached to a block of IMEM and controlled by the CP.

PMEM  Program Memory. The EEPROM (and corresponding address space) holding executable program code for the IMAPCAR2.


PRAM  Parallel Random Access Memory. A model for analyzing parallel computing architectures, and a precursor to CTA.

PU  Processing Unit. A secondary processor core available when the IMAPCAR2 is operating in MIMD Mode or Mixed Mode, created by combining four PEs.

RISC  Reduced Instruction Set Computer. A CPU design strategy focusing on a relatively small set of simple, but quickly executed instructions.

ROI  Region of Interest. A sub-region of an image that is of particular interest to analysis software, e.g., a region that may contain an obstacle and needs further processing to determine whether it does or not.

RTR  Ready To Receive. One of the handshake messages used by AutoPilot’s C-ring flood avoidance.

RTS  Ready To Send. One of the handshake messages used by AutoPilot’s C-ring flood avoidance.

SDK  Software Development Kit. The set of documentation (including specifications and example programs) and tools (including compilers and debuggers) used to develop software for a specific platform.
SIMD  Single Instruction, Multiple Data. A form of parallel processing in which each processor is operating on a different piece of data, but all must be executing the same instruction. Contrast MIMD.

SIMD Mode  An IMAPCAR2 execution mode in which no PUs are active, and the full PE array is available to the CP. In this mode it acts as a single-core processor with high-performance SIMD instructions.


Speedup  A measure of how much faster a parallel algorithm is relative to an equivalent sequential one. Calculated as $S_p = \frac{T_1}{T_p}$, where $p$ is the number of processors, $S_p$ is the speedup with $p$ processors, $T_1$ is the execution time of the sequential algorithm, and $T_p$ is the execution time of the parallel algorithm with $p$ processors.

TCL  Tool Command Language. A programming language used to implement some tools in the IMAPCAR2 SDK (and widely used elsewhere).

XC Core  Extensible Computing Core. The internal name for the architecture used in the IMAPCAR2; a distinction is made between the XC Core and the IMAPCAR2 as it may someday be used in other, related processors with different intended use cases.

XTC  Extended C. A C variant with SIMD extensions used to program the Xetal II chip and its descendants.
Appendix A

AutoPilot API Reference

This appendix contains the API reference for AutoPilot. The underlying concepts are addressed in Section 2.1; this appendix serves as a reference for individual functions and symbols in the AutoPilot API.

A.1 Types and Symbols

These are new typedefs and global constants introduced by AutoPilot. Some globals may be implemented as macros.

A.1.1 PI_WORK_FUNC

typedef void (*PI_WORK_FUNC)(int32, void*);

Function prototype for user defined worker functions.

Functions to be assigned to processes must match this prototype. The two arguments are supplied via PI_CreateProcess.

A.1.2 PI_PROCESS

The AutoPilot process type. This represents a Pilot process, executing a function of type PI_WORK_FUNC on an active PU. This is also the type of the process running on the CP (see PI_MAIN).
A.1.3 PI_MAIN

extern PI_PROCESS PI_MAIN;

The PI_PROCESS handle for the main process, i.e., the one running on the CP.

A.1.4 PI_CHANNEL

The AutoPilot channel type. This represents a unidirectional line of communication between two processes, created by calling PI_CreateChannel. It can be used to send and receive data in the execution phase using PI_Write and PI_Read, or made part of a bundle in the configuration phase using PI_CreateBundle.

A.1.5 PI_BUNDLE

The AutoPilot bundle type. This represents a collection of channels, sharing a single common endpoint at either the reader or writer end, which can be used for collective operations such as PI_Select and PI_Broadcast.

A.1.6 PI_BUNDLETYPE

typedef enum { PI_BUNDLE_READER, PI_BUNDLE_WRITER } PI_BUNDLETYPE;

Specifies which type of bundle to create — specifically, whether the common endpoint is a reader or a writer.

This is essentially a safety check; it does not restrict the use of the bundle (for select, broadcast, etc.) the way it does in MPI Pilot where the bundle type has to be fixed at creation time.

See PI_CreateBundle.
A.1.7  PI_COPYTYPE

typedef enum { PI_COPYSAME, PI_COPYREVERSE };

Specifies which direction the channels should point after a copy operation made via
PI_CopyChannels — the same as the array being copied, or in the opposite direction.

A.1.8  emem_ptr_t

typedef uint32 emem_ptr_t;

The type for physical (i.e., page-independent) memory addresses. Used for functions
such as ROI copies that bypass the normal memory access mechanisms. This is the type
associated with the %p format in PI_Write and other communication functions.

Normal memory access is 24-bit, but the IMAPCAR2’s entire address space is 28-bit,
which makes uint32 the smallest type that can safely handle physical pointers.

A.2  Configuration Phase

A.2.1  PI_Configure

uint16 PI_Configure( uint16 nprocs, size_t dmemsize,
                        uint8 * dmemptr );

Initializes the Pilot library and puts the system into MIMD mode.

This is the starting point of all Pilot code; calling PI_Configure is necessary before
any other Pilot functions can be used. It must be called from the CP when Pilot is in SIMD
mode, and will put the system in MIMD mode, activating at least nprocs PUs. It returns
the number of active PUs (which is equal to the number of processes that can be created by
calling PI_CreateProcess).
\textit{dmemsize} is the amount of data memory (heap + stack) to allocate for each PU. If \textit{dmemptr} is NULL, DMEM will be allocated automatically by AutoPilot. If it is, the memory region pointed to by \textit{dmemptr} will be partitioned into \textit{nprocs} areas of \textit{dmemsize} bytes each, which will be used as PU DMEM.

It is necessary that PU DMEM areas be aligned to a 1024-byte boundary. If \textit{dmemptr} does not satisfy this constraint, \textit{PIConfigure} will start partitioning at the next 1024-byte boundary after \textit{dmemptr}. For this reason, if you cannot guarantee that \textit{dmemptr} is correctly aligned, you should allocate \textit{nprocs \cdot dmemsize} + 1024 bytes for DMEM so that \textit{PIConfigure} does not go past the end of the allocated region.

**A.2.2 PI_CreateProcess**

\[
P I \_ \_ C r e a t e P r o c e s s ( P I \_ W O R K \_ F U N C \ f , u i n t 32 \ i n d e x , \ v o i d \ * \ p o i n t e r ) ;
\]

Creates a new process.

The next available active PU is assigned the work function \textit{f} and work arguments \textit{index} and \textit{pointer}. \textit{PICreateProcess} returns a \texttt{PI\_PROCESS} handle representing this process, which can be used with \texttt{PI\_CreateChannel}. The work function will not actually begin execution until \texttt{PI\_StartAll} is called, at which point \textit{f(index, pointer)} will be called on the corresponding PU. \textit{index} and \textit{pointer} are arbitrary values which can be used for any purpose by the programmer; it is common to use \textit{index} for a unique worker ID (allowing different workers running the same \textit{f} to discriminate among each other) and \textit{pointer} for process-specific storage. Within the function, the usual warnings about using pointers to shared memory apply.
A.2.3  PI_CreateChannel

    PI_CHANNEL PI_CreateChannel( PI_PROCESS from, PI_PROCESS to );

Creates a new channel between the specified processes. Returns a PI_CHANNEL handle representing the new channel.

Channels are unidirectional; messages can only be sent from from and to to. If bidirectional communication is required, a second channel must be created in the opposite direction.

If the channel is stored in an array, it can later be used as a bundle component with PI_CreateBundle.

A.2.4  PI_CreateBundle

    PI_BUNDLE PI_CreateBundle( PI_BUNDLETYPE type,
                              PI_CHANNEL const channels[],
                              size_t size );

Creates a channel grouping (bundle) for collective operations, and returns a bundle handle. The channels must all share a single common endpoint at either the reading or writing end.

The bundle can be used during the execution phase for collective operations. type indicates the directionality of the bundle — whether the common endpoint is a reader or writer. Read bundles can be used for PI_Select and PI_TrySelect; write bundles for PI_Broadcast. channels is the array of channels to incorporate into the bundle, and size the number of channels in the array.

Each channel can be part of only one bundle.

type is currently used for runtime checking to ensure that the bundle has the directionality expected (and thus, if an error is made, report this during configuration rather than
when the bundle is later used). It may be removed in a future version.

A.2.5 PI_CopyChannels

    PI_CHANNEL * PI_CopyChannels( PI_COPYTYPE direction,
        PI_CHANNEL const src[],
        PI_CHANNEL dst[],
        size_t size);

Copies an array of channels, optionally reversing them.

Given an array src of size channels, duplicates it — i.e., for each channel in src, it calls PI_CreateChannel on the same endpoints and stores the result at the same index in dst.

The directionality of the new channels is determined by direction: PI_COPY_SAME creates the new channels with the same directionality, while PI_COPY_REVERSE reverses them.

Returns dst.

Note that this function will not copy a bundle. If you need a copy of a bundle, use this function to copy the channels that make up the bundle and then call PI_CreateBundle on the copy.

A.2.6 PI_StartAll

    void PI_StartAll();

Ends the configuration phase and begins parallel processing.

This call begins the execution phase and activates all processes created using PI_CreateProcess. When it is called, all PUs that have been assigned an AutoPilot process will call the user function that was specified when PI_CreateProcess was called, and all PUs that have not been assigned a process will shut down.

No more channels, processes, or bundles may be created once this function is called.
A.3 Execution Phase

These functions are used during the execution phase — the actual parallel processing stage, after `PI_StartAll` has been called. During this phase, no new processes, channels, or bundles can be created, but messages can be passed freely between processes using those channels and bundles that were created during the configuration phase.

A.3.1 PI_ChannelHasData

```c
uint8 PI_ChannelHasData( PI_CHANNEL c );
```

Indicates whether a channel can be read from without blocking.

AutoPilot can determine whether a channel is ready to read from without actually reading any pending message on the channel. This information is made accessible via `PI_ChannelHasData`. If it returns 0, there is no pending message and a call to `PI_Read` on the channel would block until there is. If it returns 1, there is a pending message and `PI_Read` would not block.

If you need to check a large number of channels to see which of them are ready for reading, consider using a read bundle and `PI_Select` instead.

A.3.2 PI_Write

```c
void PI_Write( PI_CHANNEL c, const char * format, ... );
```

Writes a number of values to the channel `c`.

The values are specified as the varargs. `format` specifies the number and type of values to write, in a manner similar to the `printf` family of functions. The supported format string codes, and their meanings, are detailed in Table 3.1.

Arrays are specified by inserting the size just before the type, e.g., "%25d" corresponds
to type int[25]. In this case, the corresponding value should be a pointer to the first
element of the array. If the size is written as *, it is obtained from the argument immediately
preceding the actual value, e.g., ("%*d", size, array).

PI_Write blocks until the message is successfully delivered to the destination.

A.3.3 PI_Read

void PI_Read( PI_CHANNEL c, const char * format, ... );

Reads a number of values from the specified channel.

The format string specifies the number and types of values, in the same manner as
PI_Write. PI_Read uses a scanf-like approach; the varargs should be pointers to the
variables to fill in with the values read.

The format string on the reader must match the format string on the writer. Non-
semantic differences such as differences in whitespace are permitted, but differences in
meaning are not. Any differences are checked for after the arguments are scanned and the
format string is completely parsed, so ("%5d") and ("%*d", 5) are considered equivalent.
The sizes must match, however, as must the declared types (even if they are equivalent in
width); "%5d" and "%6d" are considered different format strings, as are "%c" and "%b".

A.3.4 PI_Select

int PI_Select( const PI_BUNDLE b );

Select the next ready-to-read channel from a bundle.

Blocks until at least one of the channels in b, which must be a reader bundle, is ready
for reading (i.e., PI_Read called on it would not block), then returns the index in the bundle
(which is also the index in the original channel array the bundle was created from) of that
channel. If multiple channels become ready for reading simultaneously, it is unspecified
which one it returns.

If you want to test if any channels in a bundle are ready for reading without blocking, use PI_TrySelect.

**A.3.5 PI_TrySelect**

```c
int PI_TrySelect( const PI_Bundle b );
```

Equivalent to PI_Select, but does not block. If there are no channels ready for reading in `b` when it is called, returns -1.

**A.3.6 PI_GetBundleSize**

```c
size_t PI_GetBundleSize( const PI_BUNDLE b );
```

Returns the number of channels in `b`.

**A.3.7 PI_GetBundleChannel**

```c
PI_CHANNEL PI_GetBundleChannel( const PI_BUNDLE b, const size_t n );
```

Returns the `n`th channel in the bundle. Channels are added the bundle in the same order that they exist in the array originally passed to PI_CreateBundle, so this is equivalent to to `n`th channel in that array.

**A.3.8 PI_Broadcast**

```c
void PI_Broadcast( PI_BUNDLE b, const char * format, ... );
```

Writes to all channels in the specified bundle simultaneously. `b` must be a write bundle.

This is equivalent to calling PI_Write on every bundle in the channel in sequence, except that it makes use of the IMAPCAR2’s internal broadcast support and is thus more
efficient than calling **PI_Write** by hand. It blocks until the messages have been delivered to all of the reader processes.

**A.3.9 PI_StopMain**

```c
void PI_StopMain();
```

Ends parallel processing and returns the system to SIMD mode.

This function can only be called by **PI_MAIN**. It blocks until all other AutoPilot processes have finished execution (i.e., they have returned from their work functions), then cleans up all Pilot data, shuts down the PUs, and switches the system back to SIMD mode. It returns only once these steps are complete.

Once the system is back in SIMD mode, no additional AutoPilot calls may be made, except for **PI_Configure** to re-activate AutoPilot and enter the configuration phase.

**A.4 Miscellaneous Functions**

These are functions that are not tied to a specific phase, and can be called in either configuration or execution phase, or even in SIMD mode.

**A.4.1 PI_StartTime**

```c
void PI_StartTime();
```

Starts an internal timer. Creates a fixed point in time — the time since **PI_StartTime** was called can be retrieved by calling **PI_EndTime**.

*Note:* Uses **IxPtime** internally, which is only valid on the CP. Consequently, this function can only be called by **PI_MAIN**.
A.4.2 PI_EndTime

double PI_EndTime();

Returns the elapsed time, in seconds, since the last call to PI_StartTime or PI_EndTime, and resets the timer.

Note: Like PI_StartTime, this function uses IxPtime and can only be called from the CP.

A.4.3 PI_Abort

void PI_Abort(const char * text, const char * file,
               const ulong line);

Aborts execution of the AutoPilot application, reports an error, and halts the processor. This may be called either by library functions when an error occurs, or explicitly by user code as a panic button. It does not perform any sort of cleanup and should be used only for unrecoverable errors.

text: The text of the error message. Ignored if logging is disabled.

file: The file in which the error occurred. Callers are recommended to use __FILE__.

line: The line at which the error occurred. Callers are recommended to use __LINE__.

Note: PUs cannot halt the entire system; if this is called on a PU, it will report the error and then raise an interrupt which causes the CP to halt the system. This may, in some cases, result in multiple error messages being reported before the system actually halts.

A.5 Build Configuration

AutoPilot is a standard 1DC library. It can be built as part of a larger project, or as a stand-alone static library which is later linked to another program. It does, however, have
compile-time configuration which affects both AutoPilot itself, and programs that include `<autopilot.h>`. This configuration is controlled by the `PI_DEBUG` and `PENO` preprocessor variables.

At `PI_DEBUG=0`, all internal safety checks are disabled. Events that would normally cause AutoPilot to abort will instead result in deadlocks or hardware exceptions. This comes with a small performance benefit, but makes debugging — or even detecting that an error has occurred — much more difficult. For this reason, the use of `PI_DEBUG=0` is discouraged. It is primarily useful for cases like microbenchmarking.

At `PI_DEBUG=1`, the default, AutoPilot’s internal error checking is enabled. The `PI_ASSERT` macro is used to check function preconditions, such as ensuring that processes don’t try to read or write channels that are not connected to them. Errors result in `PI_Abort` being called and reporting the location and nature of the error. This imposes a minor performance penalty but places most of the burden of error checking on AutoPilot itself rather than on application code. Neither `PI_DEBUG=0` nor `PI_DEBUG=1` have any effect on application code.

At `PI_DEBUG=2`, call site tracing is enabled. This causes every call to the `PI_` API to be recorded internally, and in case of error, `PI_Abort` will report not only the location of the error within Pilot itself, but also the location of the last call from application code into AutoPilot (excepting `PI_Abort` itself). This eases debugging, but imposes an additional performance cost, as every call to a `PI_` function must additionally record the file and line number of the call site. This requires both the library and any code calling it to be compiled with this setting: if it is set in AutoPilot but not in application code, tracing will not function correctly, and if set in the application but not AutoPilot, the program will not link.

`PI_DEBUG=3` enables full internal logging and is primarily of use only in debugging AutoPilot itself. It causes a vast amount of information on AutoPilot’s internal operations to
be output to the simulator or debugger console. This imposes a performance cost such that versions of AutoPilot built with this option are useable only for debugging. In application code, this setting is equivalent to PI_DEBUG=2.

PENO is a preprocessor variable set by the IMAPCAR2 SDK build process to the number of PEs available on the target platform; it does not normally need to be set by hand. It is, however, important to ensure that the value of PENO be identical across all components of the software being compiled — application code, AutoPilot, and other libraries — or the link will fail.
Appendix B

MISRA-C Compliance

MISRA-C [40] is a standard for the use of the C programming language in automotive applications. A complete MISRA-C compliance matrix covering all 142 rules in the standard is beyond the scope of this thesis; this appendix documents only those MISRA-C rules that are violated by the AutoPilot library, and why.

1.1: All code shall conform to ANSI C89. The IMAPCAR2 is programmed in 1-Dimensional C, an ANSI C89 variant; without using its specific extensions, some features of the chip are not available.

3.4 All uses of #pragma shall be documented and explained. #pragma is used in 1DC to control a number of commonly used features, including controlling whether a given function is compiled for the CP or PUs and whether or not a function is an interrupt handler or not. By their nature, isolation of these #pragmas to specific functions it not possible, and they are fully documented in the SDK documentation.

3.6 All libraries used in production code shall be written to MISRA-C compliance. Validating the IMAPCAR2 standard libraries for MISRA-C compliance is a major undertaking outside the scope of this project.
14.6-14.7 Loops shall contain at most one break; functions shall contain at most one return. Single-point-of-exit and similar structures are a relic of the early days of structured programming, and often serve only to needlessly complicate code and make it less readable. These conventions are followed in AutoPilot only when they do not hamper readability.

16.1 Functions shall not be defined with a variable number of arguments. The Pilot API requires the use of varargs functions for PI_Write and related message passing functions.

18.4 Unions shall not be used. Unions are used by the format string parser to implement “variant types” as discussed in the MISRA-C standards document.

19.x Restrictions on the use of macros. Several of these rules are violated by the macros used to implement call site tracing, which must transparently wrap calls to the AutoPilot API in order to function.

20.4 Dynamic heap memory allocation shall not be used. AutoPilot must allocate heap memory for PU DMEM in cases where the user has not specified a memory region; statically allocating such regions is not feasible, as the amount of memory needed cannot be predicted in advance. This can be avoided by statically allocating DMEM in user code and passing this DMEM region to PI_Configure, in which case AutoPilot will not perform any dynamic allocation.
Appendix C

Benchmark Code

This appendix contains the source code for the programs used to collect the benchmarking results in Section 5.2. Included are four benchmarks: mode switching, one-to-one messages, broadcast messages, and messages requiring the transfer of large buffers. These benchmarks produce output suitable for consumption by R [14] or any compatible program.
C.1 Mode Switching

```c
#include <autopilot.h>
#include <stdimap.h>

ulong t;

/* allocate slightly more than we need in case we need to re-align it */
/* "align" keyword can't be used here as it only aligns to 512 */
uint8 dmem_buf[33792];

/* address and pointer to *aligned* DMEM */
ulong dmem_addr;
void * dmem_ptr;

#pragma Kmp=1
void pilot_worker(int32 index, void * udata) {}
void lib1dc_worker(void * udata) {}
#pragma Kmp=0

/* 16 LOC */
void test_start(ushort puno)
{
    ulong dstklm = dmem_addr;
    ushort pu;
    IpTime();
    if (puno > PUNO/2)
        Ip_start_mode(IP_MP_MODE);
    else
        Ip_start_mode(IP_MIXED_MODE);
    for (pu = 0; pu < puno; ++pu)
    {
        Ip_start_pu(pu,
                    lib1dc_worker,
                    dstklm,
                    1024,
                    NULL);
        dstklm += 1024L;
    }
    Ip_join(0, 0);
    Ip_start_mode(IP_SIMD_MODE);
    t = IpTime()/133L;
    printf("%d\tIx_start_pu\t%d\n", puno, t);
}

/* 11 LOC */
void test_start_multi(ushort puno)
{
    if (puno >= 32)
    {
        printf("%d\tIx_start_multi_pu\tNA\n", puno);
        return;
    }
```
IxPtime();

if (puno > PUNO/2)
   Ix_start_mode(IX_MP_MODE);
else
   Ix_start_mode(IX_MIXED_MODE);

Ix_start_multi_pu((1L << (uint32)puno) - 1L,
   libic_worker,
   NULL, 0,
   NULL, 0,
   dmem_addr,
   1024,
   NULL);

Ix_join(0, 0);
Ix_start_mode(IX_SIMD_MODE);

t = IxPtime() / 133L;
printf("%d\tIx_start_multi_pu\t%d\n", puno, t);
}

/** 6 LLLOC */
void test_configure(ushort puno)
{
   ushort i;
   IxPtime();
   PI_Configure(puno, 1024, dmem_ptr);
   for (i = 0; i < puno; ++i)
      PI_CreateProcess(pilot_worker, 0, NULL);
   PI_StartAll();
   PI_StopMain();
   t = IxPtime() / 133L;
   printf("%d\tPI_Configure\t%d\n", puno, t);
}

void main()
{
   ushort i;
   /** PU DMEM areas need to be 1024 aligned */
   dmem_addr = ((uint32)dmem_buf + 1024L) & ~0x3FFL;
   dmem_ptr = (void*)dmem_addr;
   printf("PUNO\tImplementation\tttime\n");
   for (i = 1; i <= PUNO; ++i)
   {
      test_start(i);
      test_start_multi(i);
      test_configure(i);
   }
}
C.2 Message Passing

```c
#include <autopilot.h>
#include <stdlib.h>

ulong t;

/* allocate slightly more than we need in case we need to re-align it */
/* "align" keyword can't be used here as it only aligns to 512 */
uint8 dmem_buf[33792];

/* address and pointer to *aligned* DMEM */
ulong dmem_addr;
void * dmem_ptr;

PI_CHANNEL toWorkers[PUNO];

#pragma Kmp=1
void pilot_worker(int32 index, void * udata) {
    int d0;
    PI_Read(toWorkers[index], "%d", &d0);
}

void lib1dc_worker(void * udata) {
    ushort i;
    ix_MSG msg;
    int d0;
    msg.hd = ix_msg_header(ix_cp_id(), 0, 0);
    ix_recv_msg(&msg, 0);
    d0 = msg.d0;
}
#pragma Kmp=0

void test_ix(ushort puno)
{
    ulong dstklm = dmem_addr;
    ushort pu;
    ix_MSG msg;
    if (puno > PUNO/2)
        IX_start_mode(IX_MP_MODE);
    else
        IX_start_mode(IX_MIXED_MODE);
    for (pu = 0; pu < puno; ++pu)
    {
        IX_start_pu(pu,
                    lib1dc_worker,
                    dstklm,
                    1024,
                    NULL);
        dstklm += 1024L;
    }
    IXPtime();
    for (pu = 0; pu < puno; ++pu)
```
{  
  msg.hd = ix_msg_header(pu, 0, 0);
  msg.d0 = 0;
  ix_send_msg(&msg, 0);
}

ix_join(0, 0);
t = IxPtime()/133L;
ix_start_mode(IX_SIMD_MODE);
printf("%d\tix_send_msg\t%d\n", puno, t);

/* 6 LLOC */
void test_pi(ushort puno)
{
  short pu;
  PI_Configure(puno, 1024, dmem_ptr);
  for (pu = 0; pu < puno; ++pu)
  {
    PI_PROCESS p = PI_CreateProcess(pilot_worker, pu, NULL);
    toWorkers[pu] = PI_CreateChannel(PI_MAIN, p);
  }
  PI_StartAll();
  IxPtime();
  for (pu = puno-1; pu >= 0; --pu)
  {
    PI_Write(toWorkers[pu], "%d", 0);
  }
  ix_join(0, 0);
t = IxPtime()/133L;
  PI_StopMain();
  printf("%d\tPI_Write\t%d\n", puno, t);
}

void main()
{
  ushort i;
  /* PU DMEM areas need to be 1024 aligned */
  dmem_addr = ((uint32)dmem_buf + 1024L) & ~0x3FFL;
  dmem_ptr = (void*)dmem_addr;
  printf("PUN0\tImplementation\tttime\n");
  for (i = 1; i <= PUN0; ++i)
  {
    test_ix(i);
    test_pi(i);
  }
}
C.3 Broadcast Messages

```c
#include <autopilot.h>
#include <stdimap.h>

ulong t;

/* allocate slightly more than we need in case we need to re-align it */
/* "align" keyword can't be used here as it only aligns to 512 */
uint8 dmem_buf[33792];

/* address and pointer to *aligned* DMEM */
ulong dmem_addr;
void * dmem_ptr;

PI_CHANNEL toWorkers[PUNO];
PI_BUNDLE bundle;

#pragma Kmp=1
void pilot_worker(int32 index, void * udata) {
    int d0;
    PI_Read(toWorkers[index], "%d", &d0);
}

void lib1d_c_worker(void * udata) {
    ix_MSG msg;
    int d0;
    msg.hd = ix_msg_header(ix_pu_id(), 0, 0);
    ix_recv_msg(&msg, 0);
    d0 = msg.d0;
}

#pragma Kmp=0

/* 16 LLOC */
void test_ix(ushort puno)
{
    ulong dstklm = dmem_addr;
    ushort pu;
    ix_MSG msg;
    if (pu == PUNO/2)
       Ix_start_mode(Ix_MP_MODE);
    else
       Ix_start_mode(Ix_MIXED_MODE);
    for (pu = 0; pu < puno; ++pu)
    {
        Ix_start_pu(pu,
                    lib1d_c_worker,
                    dstklm,
                    1024,
                    NULL);
        dstklm += 1024L;
    }
    IxPtime();
}
```
msg.hd = ix_msg_header(puno-1, 0, 1);
msg.d0 = 0;
ix_send_msg(&msg, 0);
ix_join(0, 0);
t = Ixptime()/133L;
ix_start_mode(IX_SIMD_MODE);
printf("%d\txix_send_msg\t%d\n", puno, t);
}
/
*/ 6 LOC */
void test_pi(ushort puno)
{
  ushort i;
  PI_Configure(puno, 1024, dmem_ptr);
  for (i = 0; i < puno; ++i)
  {
    PI_PROCESS p = PI_CreateProcess(pilot_worker, i, NULL);
    toWorkers[i] = PI_CreateChannel(PI_MAIN, p);
  }
bundle = PI_CreateBundle(PI_BUNDLE_WRITER, toWorkers, puno);
PI_StartAll();
IxPtime();
PI_Broadcast(bundle, "%d", 0);
ix_join(0, 0);
t = IxPtime()/133L;
PI_StopMain();
printf("%d\tPI_Broadcast\t%d\n", puno, t);
}
void main()
{
  ushort i;
  /* PU DMEM areas need to be 1024 aligned */
dmem_addr = ((uint32)dmem_buf + 1024L) & ~0x3FFL;
dmem_ptr = (void*)dmem_addr;
printf("PUN0\tImplementation\tttime\n");
  for (i = 2; i <= PUN0; ++i)
  {
    test_ix(i);
    test_pi(i);
  }
}
C.4 Large Buffers

```c
#include <autopilot.h>
#include <stdimap.h>

ulong t;

/* allocate slightly more than we need in case we need to re-align it */
/* "align" keyword can't be used here as it only aligns to 512 */
uint8 dmem_buf[33792];
/* address and pointer to *aligned* DMEM */
ulong dmem_addr;
void * dmem_ptr;
PI_CHANNEL toWorker;

const char * formats[] = {
    "%lu",
    "%2lu",
    "%4lu",
    "%8lu",
    "%16lu",
    "%32lu",
    "%64lu",
    "%128lu",
    "%256lu",
    "%512lu"
};

#pragma Kmp=1
void pilot_worker(int32 size, void * udata) {
    ulong buf[512];
    PI_Read(toWorker, formats[size], buf);
}

void lib1dcd_worker(void * udata) {
    ulong buf[512];
    ulong size = (ulong) udata;
    ix_recv_msgblk(buf, 1<<size, 0);
}
#pragma Kmp=0

/* 16 LLOC */
/* 16 LLOC */
void test_ix(ushort size) {
    ulong buf[512];
    IX_start_mode(IX_MIXED_MODE);
    IX_start_pu(0,
               lib1dcd_worker,
               dmem_addr,
               4096,
               (void*)size);
    IXPtime();

    return;
}
```

ix_send_msg_blk121(buf, 1<<size, 0, 0);
Ix_join(0, 0);
t = IxTime()/133L;
Ix_start_mode(Ix_SIMD_MODE);
printf("%d\tix_send_msg_blk121\t%d\n", size, t);
}

/* 6 LLC */
void test_pi(ushort size)
{
    PI_PROCESS p;
    ulong buf[512];
    PI_Configure(1, 4096, dmem_ptr);
    p = PI_CreateProcess(pilot_worker, size, NULL);
toWorker = PI_CreateChannel(PI_MAIN, p);
PI_StartAll();
IxTime();
PI_Write(toWorker, formats[size], buf);
Ix_join(0, 0);
t = IxTime()/133L;
PI_StopMain();
printf("%d\tPI_Write\t%d\n", size, t);
}

void main()
{
    ushort i;
    /* PU DMEM areas need to be 1024 aligned */
dmem_addr = ((uint32)dmem_buf + 1024L) & ~0x3FF;
dmem_ptr = (void*)dmem_addr;
printf("size\tImplementation\tttime\n");
for (i = 0; i <= 9; ++i)
{
    test_ix(i);
    test_pi(i);
}
Appendix D

N-Queens Code

This appendix contains code for the various N-Queens implementations discussed in Section 5.3. Included are five implementations: the sequential implementation used as a reference point (D.1), the SIMD (D.2) and MIMD (D.3) versions included with the SDK, and the round-robin (D.4.1) and non-RR (D.4.2) AutoPilot implementations.

The AutoPilot versions were created by "Pilotizing" (i.e., adapting to the AutoPilot API) the MIMD code: the entire program was copied, and AutoPilot equivalents of SDK message-passing invocations were inserted. The original MIMD code has been retained in comments, with the effect that the AutoPilot version appears longer than the MIMD version, but this is largely due to extra comment lines. See the table below for lines of executable code.

Furthermore, for the purpose of calculating parallel speedup, a sequential version (D.1) was overlaid in the AutoPilot code through conditional compilation. This technique was copied from the MIMD version. When the symbol SEQ is defined for compilation, the algorithm runs on the CP with no involvement of PUs. This timing gives the sequential baseline against which the MIMD and AutoPilot parallel versions are compared.

Changing the size of the board, the number of available PUs to use, and, for the AutoPilot version, the work allocation method (round-robin or first available PU) requires re-compiling with corresponding symbol values. Execution correctness is judged by whether a run prints out the right number of solutions for a particular size board.
Table D.1: N-Queens Source Size

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Code Lines</th>
<th>Comment Lines</th>
<th>Blank Lines</th>
<th>Total Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>44</td>
<td>24</td>
<td>13</td>
<td>81</td>
</tr>
<tr>
<td>SIMD</td>
<td>69</td>
<td>30</td>
<td>26</td>
<td>125</td>
</tr>
<tr>
<td>MIMD</td>
<td>128</td>
<td>37</td>
<td>40</td>
<td>205</td>
</tr>
<tr>
<td>AutoPilot (RR)</td>
<td>109</td>
<td>66</td>
<td>36</td>
<td>211</td>
</tr>
<tr>
<td>AutoPilot (non-RR)</td>
<td>119</td>
<td>46</td>
<td>36</td>
<td>201</td>
</tr>
</tbody>
</table>

Note that in all non-SIMD versions, the heart of the computation is done in the function nq_seq(). The sequential version simply calls nq_seq() from the main() function. The MIMD and AutoPilot versions arrange for PUs to call nq_seq() with arguments to compute a subproblem as allocated by main().
D.1 Sequential Version

As explained above, this version is derived from the AutoPilot version by defining the symbol SEQ at compile time. The listing below has been edited to show only the lines resulting from defining SEQ.

```c
/* FILENAME: nqueen_pilot.lc */
/* Original code: C:\1DC\sample\mp\step2\nqueen_para.lc */
/* PILOTIZED on 24-Mar-12, W. Gardner */
/* :mode=c: */

* NOTE: 1DC binaries are PENO-specific. To change PENO, rebuild AutoPilot as well!
*
*/

#include <stdlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <autopilot.h>

#ifndef N
#define N 10 // NxN sized board
#endif

#ifndef USEPUS
#define USEPUS 0 // max. PUs to use (0=all available)
#endif

#define FIRST_TIME -1
#define MSG_ID 0x0001
#define PU_DSIZE 8192L // data size for each PU, better to be a multiple of PU cache line size
#define SEQ

double Ptime = 0; // seconds in floating point being used to prevent overflow.

outside ulong osolution[II_MAX_TOTAL_PU_NUMBER+1];

/*PILOT: Nothing was changed in recursive nq_seq; it's invoked by nq_para or main*/
#pragma Kmp=1
void nq_seq(int* a, int *b, int *c, int i, ulong *solution)
{
```
```c
int j;
ulong s;

for (j = 0; j < N; j++)
    if (a[j] && b[i+j] && c[i-j+N-1]) {
        if (i < N - 1) {
            a[j] = b[i+j] = c[i-j+N-1] = 0;
            nq_seq(a,b,c,i+1, solution);
            a[j] = b[i+j] = c[i-j+N-1] = 1;
        } else {
            (*solution)++;
        }
    }
}

// collect processing time to prevent overflow
if (N>12 && (*solution & 0xffff)==0xffff)
    Ptime += lxPtime();
}

#pragma Kmp=0
ulong solution = 0;
int main()
{
    int i, a[N], b[2*N], c[2*N];
    PI_StartTime(); // Starts processing time measurement
    // Perform N-queen sequentially using only the CP
    for (i=0;i<N;i++) a[i]=1;
    for (i=0;i<2*N;i++) b[i]=c[i]=1;
    nq_seq(a,b,c,0,&solution);

    // Show results and the processing time
    Ptime = PI_EndTime() * 1000; // convert time to ms.
    printf("N=%d, solutions=%lu\n", N, solution);
    printf("proc time=%lu us at 133MHz\n", Ptime*75/10000L);
    printf("proc time=%f msec\n", Ptime);
}
```
D.2 SIMD Version

This listing shows original source code included with the SDK, and contains special IDC SIMD operators. As stated in the comment header, the N-queens problem is not a good fit for SIMD organization, so this version was just created for comparison to a MIMD solution.

```c
/* Way to compile and create IX file from dos prompt:
   > cc idc5 -O -db1 -DN=8 THISFILE.lc -o THISFILE.ix [CR]
     (Use PU to perform N-queen, where N is 8, where N can be changed)
   > cc idc5 -O -db1 -DSEQ -DN=8 THISFILE.lc -o THISFILE.ix [CR]
     (Use only CP to perform N-queen)

   * Way to execute from dos prompt:
   > runix THISFILE.ix [CR] // uses simulator
   > runix -db THISFILE.ix [CR] // uses real machine

   * Other notes (behavior of this sample program):
     CP calculates the solution for the first two rows (#=2).
     Then, each PE is assigned to continue the remaining works.
     After PENO PEs are assigned tasks, starts running the whole PE array.
     If the number of tasks is not a multiple of PENO, starts the PE array
     just one time suppressing the activity of PEs which are not assigned
     tasks.

     The solutions found by PEs are collected to CP using IsAddsepl function.

     Actually N-queen problem is not suitable to be implemented in SIMD mode.
     This sample program is used to compare its performance with a version
     which is implemented using HP mode ( refer to ../HP/nqueen.lc ).

----------------------------------------*/
#include <stdmap.h>
#include <stdio.h>
#include <stdlib.h>
#define N 8

float Ptime = 0;

void nq_sep(sep int* a, sep int* b, sep int* c, sep int* i, sep ulong
   *solution)
{
   int j;
   sep bool;
   // The loop itself is controlled by CP
```
for (j = 0; j < N; j++) {
    // If no PE should continue, then skip
    bool = (a[j] && b[i + j] && c[i - j + N - 1] && i < N);
    if (! || bool) continue;
    mif (bool) {
        mif (i < N - 1) {
            a[j] = b[i + j] = c[i - j + N - 1] = 0;
            nq_sep(a, b, c, i + 1, solution);
        } else {
            (*solution)++;
        }
    }
}

ulong solution = 0;
int main() {
    int a[N], b[2 * N], c[2 * N];
    int i, j0, j1;
    int pe = 0;
    sep int sa[N], sb[2 * N], sc[2 * N], si;
    sep ulong sep_solution = 0;
    print_time();
    for (i = 0; i < N; i++) a[i] = 1;
    for (i = 0; i < 2 * N; i++) b[i] = c[i] = 1;
    for (i = 0; i < N; i++) sa[i] = 1;
    for (i = 0; i < 2 * N; i++) sb[i] = sc[i] = 1;
    si = N;
    for (j0 = 0; j0 < N; j0++) {
        if (a[j0] && b[0 + j0] && c[0 - j0 + N - 1]) {
            a[j0] = b[0 + j0] = c[0 - j0 + N - 1] = 0;
            for (j1 = 0; j1 < N; j1++) {
                // Distribute works to each PE
                if (a[j1] && b[1 + j1] && c[1 - j1 + N - 1]) {
                    mif (pe == _PENUM) {
                        sa[j0] = sb[0 + j0] = sc[0 - j0 + N - 1] = 0;
                        sa[j1] = sb[1 + j1] = sc[1 - j1 + N - 1] = 0;
                        si = 2;
                    }
                    // Check if all PEs have been assigned their work
                    if (++pe == _PENU) {
                        // Perform the rest of works using the PE array.
                        nq_sep(sa, sb, sc, 2, &sep_solution);
                    }
                }
            }
        }
    }
    for (i = 0; i < N; i++) sa[i] = 1;
    for (i = 0; i < 2 * N; i++) sb[i] = sc[i] = 1;
    si = N;
}

a[j0] = b[0 + j0] = c[0 - j0 + N - 1] = 1;

} 

} 

// if number of tasks is not a multiple of PEND, start PE array here again.
// Value of "si" will be differ between PEs: PEs whose "si" is N will not contribute.
if (pe != 0)
    iq_sep(sa, sb, sc, si, &sep_solution);

// collect all solutions over PEs
solution = IxAddsep1(sep_solution);

// Show result and processing time
Ptime += IxPtime();
printf("N=%d, solutions=%lu\n", N, solution);
//printf("proc time=%luus at 133MHz\n", Ptime*75/L00000);
printf("proc time=%2.2fus at 133MHz\n", Ptime*75.0/10000.0);
}
D.3 MIMD Version

This listing shows original source code included with the SDK, modified slightly to support running with PU numbers other than the maximum (to allow collection of performance data for all possible PU counts). Its sequential version is compiled from it by setting the SEQ symbol. However, comparisons of MIMD parallel times were made against the C.1 sequential code (practically identical to this sequential version, but using AutoPilot timing functions PI_StartTime and PI_EndTime rather than IXpTime).

```c
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
#include <stdio.h>
#include <stdlib.h>

#if !defined N
#define N 10 /* N x N sized board */
#endif

#if !defined USEPUS
#define USEPUS 32
#endif

#define FIRST_TIME -1
#define MSG_ID 0x0001
#define PU_DSIZE 8192L /* data size for each PU, better to be a multiple of PU cache line size */

# Way to compile and create IX file from dos prompt:
> cc1dc5 -O -db1 -D#N=8 THISFILE.lc -o THISFILE.ix [CR]
(Use PU to perform N-queen, where N is 8, where N can be changed)

> cc1dc5 -O -db1 -DSEQ -D#N=8 THISFILE.lc -o THISFILE.ix [CR]
(Use only CP to perform N-queen)

# Way to execute from dos prompt:
> runix THISFILE.ix [CR] // uses simulator
> runix -db THISFILE.ix [CR] // uses real machine

# Other notes (behavior of this sample program):
CP calculates the solution for the first two rows (N x 2).
Then, each PU is asked to continue the remaining work.
The solutions found by PUs are passed to CP via outside array "osolution"
When using real machine, if printf() is executed within PU tasks,
the displayed processing time will become quite large.
If SEQ is defined, then CP perform all tasks by itself.
*/
```

163
float Ptime = 0; // floating point being used to prevent overflow.
outside ulong osolution[IX_MAX_TOTAL_PU_NUMBER+1];

#pragma Kmp=1
void nq_seq(int* a, int *b, int *c, int i, ulong *solution)
{
    int j;
    ulong s;
    for (j = 0; j < N; j++)
        if (a[j] & b[i + j] & c[i - j + N - 1]) {
            if (i < N - 1) {
                a[j] = b[i + j] = c[i - j + N - 1] = 0;
                nq_seq(a,b,c,i+1,solution);
                a[j] = b[i + j] = c[i - j + N - 1] = 1;
            } else {
                (*solution)++;
            }
        }
}
#endif

#pragma Kmp=1
void nq_para()
{
    int a[N], b[2 * N], c[2 * N];
    int i,j0,j1=0;
    ulong solution=0;
    int pduid = ix_pu_id();
    ix_MSG msg;
    while (1) {
        // Waiting for message from CP
        msg.hd = ix_msg_header(puid,MSG_ID,0);
        ix_recv_msg(&msg,0);
        #ifdef DEBUG
        printf("PU%d received=0x%x
",puid,msg.d0);
        #endif
    if (msg.d0==N) break;
    // Initialize working area
    for (i=0;i<N;i++) a[i]=1;
    for (i=0;i<2*N;i++) b[i]=c[i]=1;
    // setup the starting condition specified by CP message
    j0 = msg.d0 & 0xff;
    j1 = msg.d0 >> 8;
    // now, calls the sequential N-queen function
    a[j0] = b[0 + j0] = c[0 - j0 + N - 1] = 0;
    a[j1] = b[1 + j1] = c[1 - j1 + N - 1] = 0;
    nq_seq(a,b,c,2,&solution);
}
#endif

// Stores founded number of solutions in an outside array
o solution[ puid ] = solution;
#endif DEBUG
printf("PU%lu(tested:0x%llx:in:total:lu:solution:found\n",puid,msg.d0,solution);
#endif
}

#pragma Kmp=0
ulong solution = 0;
int main()
{
    int i, a[N], b[2 * N], c[2 * N];
#endif SEQ // SEQUENTIAL (CP) IMPLEMENTATION
IxPtime(); // Starts processing time measurement
// Perform N-queen sequentially using only the CP
for (i=0;i<N;i++) a[i]=i;
for (i=0;i<2*N;i++) b[i]=c[i]=1;
q_seq(a,b,c,0,&solution);
#else // PARALLEL IMPLEMENTATION
uint j0, j1;
ulong puno = (1L << (uint32)USEPUS) - 1L; // use all PUs
ulong pu_dstklm = (ulong)malloc(PU_DSIZE*USEPUS); // space for each PU
ix_MSG msg;
ulong pf_func[10];
if (!puno) puno = 0xFFFFFFFF;
IxPtime(); // Starts processing time measurement
// Start PUs
ix_start_multi_pu(puno, // activate P0 and P1
    q_seq, // all PUs start from here
    NULL,
    0,
    NULL,
    0,
    pu_dstklm,
    PU_DSIZE,
    NULL);
Ix_initm(); // Cleanup C-ring (just for sure)
// Initialize working area
for (i=0;i<N;i++) a[i]=i;
for (i=0;i<2*N;i++) b[i]=c[i]=1;
// Create message header which will be used throughout the followin loop
msg.hd = ix_msg_header_sync(0, MSG_ID, 1);
// Calculate the available positions in the first and second row,
// and distribute each of them as initial starting point for PUs to continue.
for (j0 = 0; j0 < N; j0++) {
    if (a[j0] && b[0 + j0] && c[0 - j0 + N - 1]) {
        a[j0] = b[0 + j0] = c[0 - j0 + N - 1] = 0;
        for (j1 = 0; j1 < N; j1++)
```c
if (a[j1] && b[1 + j1] && c[1 - j1 + N - 1]) {
  #ifdef DEBUG
  printf("taskU(j0=%d,j1=%d)\n",j0,j1);
  #endif
  msg.d0 = (ulong)((j1<<8) + j0);
  ix_send_msg(&msg,0);
  ix_recv(ix pu id(),MSG_ID,0);
}

a[j0] = b[0 + j0] = c[0 - j0 + N - 1] = 1;
}

// Terminate all PUs (as all tasks have been distributed)
// Only PUs which has finished its task will receive this broadcast message.
// This message will disappear after been received "ix_total pu number()" times.
msg.hd = ix_msg_header(USEPUS-1,MSG_ID,1);
msg.d0 = N;
ix_send_msg(&msg,0);

// Make sure that all PUs are terminated
ix_join(0,0);
free((void*)pu_dstklm);

// Collect results:
// osolution is then "outside" array used for CP-PU communication.
for (i=0; i<USEPUS; i++)
  if ((puno>>i)&1) {
    #ifdef DEBUG
    printf("osolution[%d]=%lu\n",i,osolution[i]);
    #endif
    solution += osolution[i];
  }

// Show results and the processing time
Ptime += IxPtime();
printf("N=%d, solutions=%lu\n", N, solution);
printf("proc time=%lus at 133 MHz\n", Ptime*1000000);
printf("proc time=%lus at 133 MHz\n", Ptime*1000000.0);
```

D.4 AutoPilot Versions

These two listings are generated from the same source file. D.4.1 is the round-robin version with symbol WORK=1. D.4.2 does opportunistic work allocation using PI_Select to find the next free worker process, compiled with WORK=2.

This is the only difference between the two versions. The non-round-robin version should be superior in terms of load-balancing, but it comes at a cost of slightly higher overhead for work allocation.

D.4.1 Round-Robin

```c
/* FILENAME: nqueen_pilot_rr.lc */
/* AutoPilot N-Queens implementation */
/* Based on the MIDE version included in the IMAPCAR2 SDK: */
/* /sample/mp/part2/nqueenpara.lc
This version uses round-robin work allocation.
*/
/* NOTE: IDC binaries are PENO-specific. To change PENO, rebuild AutoPilot as well! */
#include <stdinimap.h>
#include <stdio.h>
#include <stdlib.h>
#include <autopilot.h>

#ifndef N
#define N 8       // N x N sized board
#endif

#ifndef USEPUS
#define USEPUS 0  // max. PUs to use (0=all available)
#endif

//PILOT: channel variables
PI_CHANNEL pwork[IX_MAX_TOTAL_PU_NUMBER], // sending work allocation to PUs
        pusoln[IX_MAX_TOTAL_PU_NUMBER];  // collecting no. of solutions from PUs

//#define FIRST_TIME -1
//#define MSG_ID 0x0001
#define PU_DSIZE 8192L  // data size for each PU, better to be a multiple of PU cache line size

double Prime = 0;  // seconds in floating point being used to prevent overflow.
outside ulong osolution[IX_MAX_TOTAL_PU_NUMBER+1];
```
//PILOT: Nothing was changed in recursive nq_seq; it's invoked by nq_para or main
#pragma Kmp=1
void nq_seq(int* a, int* b, int* c, int i, ulong *solution)
{
    int j;
    ulong s;

    for (j = 0; j < N; j++)
    {
        if (a[j] && b[i + j] && c[i - j + N - 1]) {
            if (i < N - 1) {
                a[j] = b[i + j] = c[i - j + N - 1] = 0;
                nq_seq(a,b,c,i+1, solution);
                a[j] = b[i + j] = c[i - j + N - 1] = 1;
            } else {
                (*solution)++;
            }
        }
    }
}

//PILOT: process function called with its puid
#pragma Kmp=1
void nq_para( int32 puid, void *dummy )
{
    int a[N], b[2 * N], c[2 * N];
    //PILOT: change j0 & j1 to uint32 since prototype only does that datatype now
    // int i, j0, j1=0;
    int i;
    uint32 j0, j1, notused;
    ulong solution; // /=0;
    // int puid = i_x_pu_id();
    // i_x_MSG msg;
    while (1) {
        // Waiting for message from CP
        // msg.hd = i_x_msg_header(puid,MSG_ID,0);
        // i_x_recv_msg(&msg,0);
        PI_Read( puwork[puid], "%lu,%lu", &j0, &j1 ); // get work allocation from PI_MAIN
        #ifdef DEBUG
        // printf("PU%d received=0x%zlu",puid,msg.d0);
        printf("PU%d received j0=%lu,j1=%lu\n",puid,j0,j1);
        #endif
        // if (msg.d0==0) break;
        if (j0==N) break;

        // Initialize working area
        for (i=0;i<N;i++) a[i]=1;
        for (i=0;i<2*N;i++) b[i]=c[i]=1;

        // setup the starting condition specified by CP message
        // j0 = msg.d0 & 0xff;
        // j1 = msg.d0 >> 8;
        // now, calls the sequential N-queen function
a[j0] = b[0 + j0] = c[0 - j0 + N - 1] = 0;
a[j1] = b[1 + j1] = c[1 - j1 + N - 1] = 0;
solution = 0;
q_seq(a,b,c,2,&solution);

//PILOT: Send no. solutions found back to PI_MAIN (lets it know we're free)
PI_Write( pusln[puid], "%lu", solution );

// Stores founded number of solutions in an outside array
osolution[puid] = solution;
//ifdef DEBUG
// printf("PU%d (tested 0x%x): in total %lu solution found\n",puid,k,solution);
//endif
#endif
#pragma Kmp=0
ulong solution = 0;
int main()
{
    int i, a[N], b[2 * N], c[2 * N];
    uint j0, j1;
    // ulong puno = 0xffffffff; // use all PUs
    // ulong pu_dstklm = (ulong)malloc(PU_DSIZE*ix_total_pu_number()); // space for each PU
    // ix_MSG msg;
    // ulong pfunc[10];

    int pustatus[IX_MAX_TOTAL_PU_NUMBER];        // 0, finished/available; 1, assigned work
    ulong pusolution[IX_MAX_TOTAL_PU_NUMBER];
    int pus = ix_total_pu_number();              // in case it's < IX_MAX_TOTAL_PU_NUMBER
    int nextpu = 0;                               // next
    // pu to send work (round robin), ranges 0..pus-1
    int parcels = 0;                              // no. of distributed work parcels
    uint32 snl,notused;

    PI_StartTime(); // Starts processing time measurement

    // Start PUs
    // In_start_multi_pu(puno, // activate PU0 and PU1
    //            q_para, // All PUs start from here
    //            NULL,
    //            0,
    //            NULL,
    //            0,
    //            pu_dstklm,
    //            PU_DSIZE,
    //            NULL);
    //
    // In_initm(); // Cleanup C-ring (just for sure)
    //PILOT: Configure Autopilot process/channel architecture
if (USEPUS > 0 && pus > USEPUS) pus = USEPUS;  // limit PUs used, for debugging purposes
PI_Configure(pus, PU_DSIZE, NULL);
for (i = 0; i < pus; ++i)
{
    PI_PROCESS pu = PI_CreateProcess(nq_para, i, NULL);
pwork[i] = PI_CreateChannel(PI_MAIN, pu);
pusoln[i] = PI_CreateChannel(pu, PI_MAIN);
}
PI_StartAll();

// ------- PI_MAIN commences here -------

// Initialize working area
for (i=0; i<N; i++) a[i]=1;
for (i=0; i<2*N; i++) b[i]=c[i]=1;

// PILOT: Initialize pu status and solution counts
for (i=0; i<IX_MAX_TOTAL_PU_NUMBER; i++)
pstatus[i] = pu_solution[i] = 0;

// Create message header which will be used throughout the following loop
// msg.hd = ix_msg_header_sync(0, MSG_ID, 1);

// Calculate the available positions in the first and second row, and distribute each of them as initial starting point for PUs to continue.
for (j0 = 0; j0 < N; j0++)
    if (a[j0] && b[0 + j0] && c[0 - j0 + N - 1])
        for (j1 = 0; j1 < N; j1++)
            if (a[j1] && b[1 + j1] && c[1 - j1 + N - 1])
                // msg.d0 = (ulong)((j1<<8) + j0);
                // ix_send_msg(&msg, 0);
                // ix_recv(ix_pu_id(), MSG_ID, 0);

#--------------------------------------------------------------------- ROUND-ROBIN ALLOCATION---------------------
if ( pstatus[nextpu] == 1 ) {  // next PU is already busy
    PI_Read( pu_soln[nextpu], "%lu", &soln );  // pick up its result
    pu_solution[nextpu] += soln;  // add to its local total
}
    PI_Write( pwork[nextpu], "%lu", (uint32)j0, (uint32)j1 );  // dispatch work allocation to next PU
    pstatus[nextpu] = 1;  // mark busy
    nextpu = (nextpu+1)%pus;
    // increment round-robin index
    parcels++;
    // increment no. of distributed work parcels
}
a[j0] = b[0 + j0] = c[0 - j0 + N - 1] = 1;
for (i=0; i<pus; i++) {
    if ( pustatus[i] == 1 ) {
        PI_Read( pusoln[i], "%lu", &soln );
        pusolution[i] += soln;          // add to its local total
    }
    PI_Write( puwork[i], "%lu\%lu", (uint32)N, 0 ); // signal to stop work
    solution += pusolution[i];
}
  }  // Show results and the processing time
  Ptime = PI_EndTime() * 1000;       // convert time to ms.
  printf("N=\%ld, solutions=\%lu\n", N, solution);
  printf("proc time=\%lu us at 133MHz\n", Ptime*75/10000L);
  printf("proc time=\%f msec\n", Ptime);
  printf("Parallel version sent\%ld parcels to\%ld PUs\n", parcels, pus);
D.4.2 Non-Round-Robin

```c
/* FILENAME: nqueen_pilot_rr.llc
 * AutoPilot N-Queens implementation
 * Based on the MIND version included in the INAPCAR2 SDK:
 * /sample/mp/part2/nqueen_para.llc
 * This version uses first-available work allocation.
 * NOTE: IDC binaries are PENO-specific. To change PENO, rebuild AutoPilot as well!
 */
#include <stdlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <autopilot.h>

#define N 8 // N x N sized board

#define USEPUS 0 // max. PUs to use (0=all available)

PI_CHANNEL pwork[IX_MAX_TOTAL_PU_NUMBER], // sending work
    pusoln[IX_MAX_TOTAL_PU_NUMBER]; // collecting no. of solutions from PUs
PI_BUNDLE alpus; // needed by first-available work allocation method

#define FIRST_TIME -1
#define MSG_ID 0x0001
#define PU_DSIZE 8192L // data size for each PU, better to be a multiple of PU cache line size
double Ptime = 0; // seconds in floating point being used to prevent overflow.

int freePU( int stat[], int len ) {
    int i;
    for ( i=0; i<len; i++ ) {
        if ( stat[i] == 0 ) return i;
    }
    return -1;
}

//PILOT: Nothing was changed in recursive nq_seq; it's invoked by nq_para or main
#pragma Kmp=1
void nq_seq(int* a, int *b, int *c, int i, ulong *solution)
{
```
```c
int j;
ulong s;

for (j = 0; j < N; j++)
  if (a[j] && b[i + j] && c[i - j + N - 1]) {
    if (i < N - 1) {
      a[j] = b[i + j] = c[i - j + N - 1] = 0;
      nq_seq(a, b, c, i+1, solution);
      a[j] = b[i + j] = c[i - j + N - 1] = 1;
    } else {
      (*solution)++;
    }
  }
}

//PILOT: process function called with its puid
#pragma Kmp=1
void nq_para( int32 puid, void *dummy )
{
  int a[N], b[2 * N], c[2 * N];
  //PILOT: change j0 & j1 to uint32 since prototype only does that datatype now
  // int i, j0, j1=0;
  int i;
  uint32 j0, j1, notused;
  ulong solution;  //=0;
  int puid = ix_pu_id();
  // ix_MSG msg;
  while (1) {
    // Waiting for message from CP
    msg.hd = ix_msg_header(puid, MSG_ID, 0);
    // ix_recv_msg(&msg,0);
    PI_Read( puwork[puid], "%lu%lu", &j0, &j1 ); // get work allocation from PI_MAIN
    #ifdef DEBUG
    // printf("PU%d received=0x%llx\n",puid,msg.d0);
    printf("PU%d received j0/j1=0x%lu/0x%lu\n",puid,j0,j1);
    #endif
    // if (msg.d0==0) break;
    if (j0==N) break;
    // Initialize working area
    for (i=0;i<N;i++) a[i]=1;
    for (i=0;i<2*N;i++) b[i]=c[i]=1;
    // setup the starting condition specified by CP message
    // j0 = msg.d0 & 0xff;
    // j1 = msg.d0 >> 8;
    // now, calls the sequential N-queen function
    a[j0] = b[0 + j0] = c[0 - j0 + N - 1] = 0;
    a[j1] = b[1 + j1] = c[1 - j1 + N - 1] = 0;
    solution = 0;
    nq_seq(a, b, c, 2, &solution);
    //PILOT: Send no. solutions found back to PI_MAIN (lets it know we're free)
    PI_Write( pusoln[puid], "%lu", solution );
  }
```
# pragma Kmp=0

```c
ulong solution = 0;

int main()
{
    int i, a[N], b[2 * N], c[2 * N];
    uint j0, j1;

    int pustatus[IX_MAX_TOTAL_PU_NUMBER];         // 0,
        finished/available; 1, assigned work
    ulong pusolution[IX_MAX_TOTAL_PU_NUMBER];
    int pus = ix_total_pu_number();               // in case
        it's < IX_MAX_TOTAL_PU_NUMBER
    int nextpu = 0;
        // next
    pu to send work (round robin), ranges 0..pus-1
    int parcels = 0;
        // no.
    of distributed work parcels
    uint32 soln, notused;
    PI_StartTime();     // Starts processing time measurement

    //PILOT: Configure Autopilot process/channel architecture
    if ( USEPUS > 0 & & pus > USEPUS ) pus = USEPUS;    // limit PUs used,
        for debugging purposes
    PI_Configure(pus, PU_DSIZE, NULL);
    for ( i = 0; i < pus; ++i)
    {
        PI_PROCESS pu = PI_CreateProcess(nq_para, i, NULL);
        puwork[i] = PI_CreateChannel(PI_MAIN, pu);
        pusoln[i] = PI_CreateChannel(pu, PI_MAIN);
    }

    allpus = PI_CreateBundle( PI_BUNDLE_READER, pusoln, pus );     // selector
        bundle
    PI_StartAll();

    -------- PI_MAIN commences here  --------

    // Initialize working area
    for ( i=0; i<N; i++) a[i]=1;
    for ( i=0; i<2*N; i++) b[i]=c[i]=1;

    //PILOT: Initialize pu status and solution counts
    for ( i=0; i<IX_MAX_TOTAL_PU_NUMBER; i++)
        pustatus[i] = pusolution[i] = 0;

    // Create message header which will be used throughout the following loop
    // msg.hd = i_msg_header_sync(0, MSG_ID, 1);

    // Calculate the available positions in the first and second row,
    // and distribute each of them as initial starting point for PUs to
```
for (j0 = 0; j0 < N; j0++) {
    if (a[j0] && b[0 + j0] && c[0 - j0 + N - 1]) {
        a[j0] = b[0 + j0] = c[0 - j0 + N - 1] = 0;
        for (j1 = 0; j1 < N; j1++) {
            if (a[j1] && b[1 + j1] && c[1 - j1 + N - 1]) {
                // msg.do = (ulong)((j1<<8)+j0);
                // ix_send_msg(&msg,0);
                // ix_recv(ix_pid(),MSG_ID,0);
                // FIRST AVAILABLE PU ALLOCATION ---------------
                if ( (nextpu = freePU( pustatus, pus )) >= 0 ) {
                    // do we have a free PU?
                    pustatus[nextpu] = 1;
                    // mark it busy
                } else {
                    nextpu = PI_Select( allpus );  // wait for
                    // first one with solution
                    PI_Read( pusoln[nextpu], "%lu", &soln );  //
                    // pick up its result
                    pusolution[nextpu] += soln;  //
                    // add to its local total
                }
                PI_Write( puwork[nextpu], "%lu,%lu", (uint32)j0,
                          (uint32)j1 );  // dispatch work allocation to
                // next PU
                parcels++;
                // increment no. of distributed work parcels
            }
        }
        a[j0] = b[0 + j0] = c[0 - j0 + N - 1] = 1;
    }
}

//PILOT: run through all PUs, getting result from busy ones, and
//signalling quit
for (i=0; i<pus; i++) {
    if ( pustatus[i] == 1 ) {
        PI_Read( pusoln[i], "%lu", &soln );  // pick up
        // its result
        pusolution[i] += soln;  // add to its local total
    }
    PI_Write( puwork[i], "%lu,%lu", (uint32)N, 0 );  // signal to stop
    // work
    solution += pusolution[i];
}

// Show results and the processing time
Ptime = PI_EndTime() * 1000;  // convert time to ms.
printf("N=%d, solutions=%lu\n", N, solution);
printf("proc time=%luus at 133MHz\n",Ptime*75/10000L);
printf("proc time=%lfms at 133MHz\n",Ptime);