Ada and cc-NUMA Architectures
What can be achieved with Ada 2005?

A.J. Wellings, A. H Malik, N.C. Audsley and A. Burns
Department of Computer Science, University of York
Heslington, York YO10 5DD, UK
(andy,haseeb,neil,burns)@cs.york.ac.uk

Abstract

Real-time systems are finding it difficult to make the shift from single processor systems to multiprocessors because of the lack of support from programming platforms for multiprocessors. Although, Ada provides some support for SMPs, it's goal is to hide the complexity of the architectures so that the programmers are not distracted by low-level architectural issues. This paper argues that programmer should be given enough visibility to use the underlying architecture predictably and efficiently. We focus on the issue of memory management and memory accesses on a cc-NUMA architecture. A cc-NUMA architecture is chosen, as we believe it to be more scalable than SMP systems.

1 Introduction

Single processor systems are reaching the physical limits of speed and a large amount of effort and cost is required for the little gain in performance [4]. As a result, single processor systems are becoming obsolete, while multiprocessors more and more prevalent. Future architectures will replace these high-clock-speed processors with many simpler and smaller heterogeneous processing elements. These new architectures will be capable of producing very high performance; however, their programming will be more complex than in the single processor case.

Real-time systems are also increasing in size and complexity. The only architectures that will be able to satisfy the needs of these applications will be multiprocessors. Unfortunately, architectural complexity and tight timing constraints make the development of real-time systems on heterogeneous multiprocessor architectures extremely difficult. The usual objective of programming platforms for these systems is to hide the complexity of the architectures so that the programmers are not distracted by low-level architectural issues. However, for real-time systems, the programmer should be given *enough* visibility to use the underlying architecture predictably and efficiently. If achieved, this will lead to programmers being more productive and at the same time predictability and performance of the applications will also increase.

Ada was designed to support multiprocessor applications where there is memory shared between each processor. It assumes that all processors are essentially identical and (caching aside) access to main memory is uniform (hence Symmetric MultiProcessor (SMP) systems). At the last workshop, the adequacy of the Ada facilities for this architecture were discussed [14, 13]. The conclusion was that Ada works well where there is application requirements for global scheduling (although some implementations may also support the permanent binding of tasks to single processors or subsets of processors).

In a Non Uniform Memory Architecture (NUMA), the access to main memory may vary considerably - although the illusion of a single address space is still provided. The goal of this paper is to present some first thoughts on what can be achieved with current Ada on NUMA, in particular cc-NUMA (cache-coherent NUMA) . Our subtext is to ensure that multiprocessor issues are kept on the agenda for the coming workshop. Section 2 briefly discusses related work in this area and summarises the facilities found in the Linux OS. In section 3 of this paper we define what we mean by NUMA and introduce the notion of a *region of coherence*. We present also an overall architecture for discussion. In section 4, we consider the Ada programming abstractions that might be useful in this environment.

2 Related Work

In the past, most of the work done to develop programming languages to support parallel architectures has come from the high performance computing (HPC) community. The HPC community has been looking to increase performance and productivity for new complex architectures. The efforts of the Defence Advanced Research Project Agency (DARPA) are most prominent in the development of new programming languages. It has launched a high productivity computing systems (HPCS) programme which funds the development of three new programming languages. These languages provide new constructs which are easy to use and optimize performance of applications for parallel machines. All these languages provide distributed loops and non blocking synchronization mechanisms for maximum concurrency.

The first programming language that is being developed under the HPCS programme is the X10 [10]. X10 uses the Java programming language as a base and provides higher level abstractions to increase programmer productivity and performance. Instead of using the traditional threads, locks and barriers, X10 introduces new constructs such as activities, asynchronous activities, atomic sections, clocks, places and futures.

The second programming language being developed under the HPCS programme is Fortress [1]. This is a general purpose programming language that focuses on the scientific community by using a syntax based on simple mathematical notations.

The third language being developed under the HPCS programme is Chapel [8]. Chapel stands for Cascade High Productivity Language. It provides an object oriented framework which is able to develop applications that can be mapped over several memory units and processing elements.

One important feature of all the above languages is that they make the locality information explicitly visible to the programmer. X10 has multiple non-overlapping *places*; these are virtual locations which group together multiple activities (threads) and objects. Physical location of places can change depending upon the load balancing policy. Activities and objects remain in the same place for their lifetime and are unable to migrate to other places; Remote access is only possible by spawning asynchronous activities which are used to communicate between two places. Fortress provides *regions* which abstractly describe the architecture of the hardware on which the application is running. Each region contains an execution level where the threads reside and a memory level where the objects reside. The programmer is able to place threads in a particular region and objects in most cases are placed local to the thread calling the constructor. Chapel provides *locales* which represent units of uniform memory access (nodes). There is no difference between local and remote access as there is in X10. Chapel also allows programmers to distribute objects into specific locales and objects are not bound to a single locale for their lifetime.

2.1 Support for cc-NUMA Systems in Linux and the ACPI Specification

The POSIX Standard has yet to address multiprocessor issues; consequently, operating systems vary in the level of support they provide. Here, we briefly discuss the facilities provided by Linux as an example.

Many changes have been made to the Linux kernel 2.6 to support multiprocessors. These changes include the introduction of per processor dispatch queues for flexible scheduling and new memory allocation mechanisms based on per node memory pools [12]. Linux provides a NUMA API [11] which includes new system calls at the kernel level to support different scheduling policies and different memory allocation policies. At the programmer level, a library is provided which enables the programmer to place a page of memory on a specific node and execute a thread on a specific CPU [12]. In terms of memory allocations, the following policies can be defined [12].

- *mbind_default* – Allocate memory on the local node (the node the thread requesting allocation currently is running on)
- *mbind_preferred* – Try to allocate on a particular node first. If this fails, allocate anywhere.

- *mbind_bind* – Allocate on a specific set of nodes. Allocation will fail if the nodes are unable to provide the memory.
- *mbind_interleave* – Spread out memory allocations over all available nodes.

Notwithstanding the Linux facilities, operating systems in general have struggled to provide the appropriate application-level support for multiprocessor. The main hurdle is the abstraction layer at the hardware level which is provided to hide the complexity of the underlying architecture. This abstraction layer not only hides the complexity but it also removes the transparency which is essential to develop predictable systems.

In order to provide some transparency, the Advanced Configuration of Power Interface (ACPI) specification 3.0 [7] has been developed to establish industry common interfaces. The goal is to enabling robust operating system (OS)-directed motherboard device configuration and power management of both devices and entire systems. The ACPI has optional support for NUMA architectures. It defines the concept of a *system locality* (or *proximity domain*) and provides information about the "distances" between them. Each proximity domain is given an integer identifier. An operating system can assume that two devices in the same proximity domain are tightly coupled, but it cannot make any assumption on the distances between the domains given only their ids. Instead, two tables are provided: the system resource affinity table (SRAT) and the system locality distance information table (SLIT). The former allows the domain of a device to be obtained and the latter the relative distance between the domains to be determined. The OS can choose to optimize its behavior based on the information in these tables. The Linux kernel parses these tables on boot up and presents this information at the application level in the form of APIs and libraries.

3 Architectural Model

The cc-NUMA architecture combines the features of distributed systems and symmetric multiprocessors. While the memory is physically distributed, cc-NUMA systems provide a global address space and coherent caches. Nodes of a cc-NUMA system are usually single processors or symmetric multiprocessors (SMPs). A processor is able to access memory on the local node faster than memory on a remote node. Examples of cc-NUMA systems range from small AMD Opteron based systems such as HP Proliant servers [9] to large and expensive SGI Altix systems [15]. In this paper we use a simple 4 node cc-NUMA system, which is based on AMD Opteron processor as shown in Figure 1, as an example architecture. Some features of this system are discussed below:

1. The AMD Opteron has four AMD dual core processors that are connected by a very high speed bus called the coherent HyperTransport (cHT). The cHT is a high speed point to point link between the processors. Each processor corresponds to a node and is given a NodeIds from 0 to 3.
2. The AMD processor can be divided into two main parts i.e. the CPU cores and the northbridge architecture. Each processor is attached to a northbridge which acts as an interface for the I/O devices and memory present in the system [6]. The northbridge consists of the following components:
   (a) System Request Interface(SRI): This takes requests from the CPU cores and forwards them to the concerned functional units.
   (b) Crossbar (XBAR): The crossbar connects all the other components in the northbridge.
   (c) Memory Controller (MCT): The memory controller provides an interface to the memory connected to the system.
   (d) HyperTransport ports: These ports allow the processor to connect to other processors via the cHT.
3. Each node has an integrated memory controller. At boot time, all nodes locate their local memory and map it onto a single physical global address space [2]. Usually the higher order bits of the physical address determine to which node the address belongs. The physical address space (PAS) is not necessarily contiguous i.e. there can be holes in the PAS.

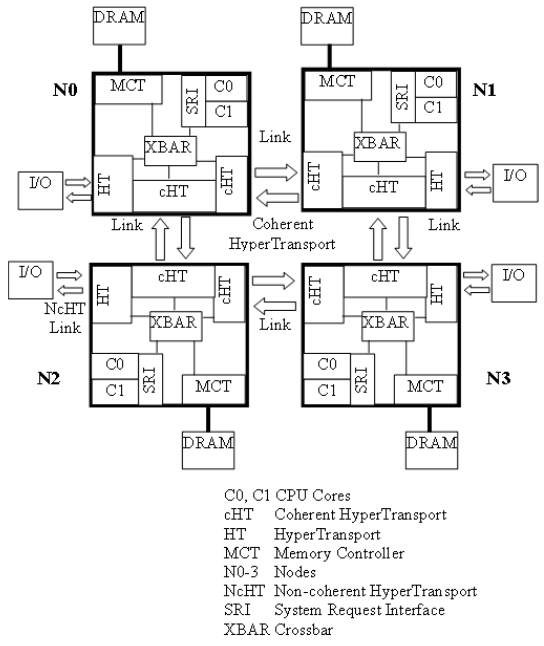

Figure 1: 4 Node cc-NUMA Architecture based on AMD Opteron taken from [3]

4. The latency of memory access depends on the location from where the memory is being accessed. It is not necessary that all the nodes are directly connected to each other; however there must be an indirect path that exists between the nodes through other nodes. Figure 2 shows the memory hierarchy of a cc-NUMA system. Although all processors will have the same view of memory, the access time will depend on the location of the data in the hierarchy, as illustrated in Figure 2.

5. In Figure 2, distance is measured as the number of hops from the processor accessing the data to the memory bank where the data resides. The inter-node distances for Figure 1 are given in Table 1.

$Distances$	$N0$	$N1$	$N2$	$N3$
N0	0	1	1	2
N1	1	0	2	1
N2	1	2	0	1
N3	2	1	1	0

Table 1: Inter-node Distances for Figure 1

6. The latency of memory accesses depends on the saturation of resources as well as the distance. [3] discusses in details how distances and saturation of resources affects the latency. Table 2 gives a simple case where latency depends on distance measured in [3]. As the value of k (distance) increases, the latency will also increase. Here a time of 100 represents the time taken for a single read operations; hence we are showing percentage increases rather than physical access times.

There are two points worth making about this example architecture:

1. The architecture physically supports a coherent memory model. By this we mean that the individual processors see the same physical memory because the local caches are kept coherent. Hence the whole memory space is a *physical region of coherence*.

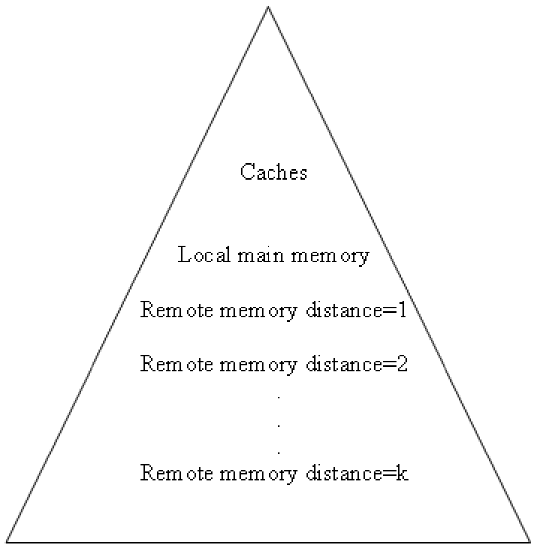

Figure 2: Memory Hierarchy of a cc-NUMA System

$Nodes$	$Read$	$Write$
N0	100	113
N1	108	127
N2	109	129
N3	130	149

Table 2: Percentage of Time taken by Different Memory Accesses in Figure 1 from N0

2. There can be significant performances increases by ensuring that when cache misses occur, all main memory accesses are local. For example, read access to memory can be 30% slower.

4 Useful Ada Abstractions for Programming cc-NUMA

Of course, an Ada compiler can ignore all NUMA issues and simply treat the machine as an SMP. However, here we assume that there are significant performance and predictability gains to be had by the compiler and run-time being cc-NUMA aware.

There are perhaps three approaches that can be taken. The first is for the compiler to try to optimize code and data placement. However, without significant data and control flow analysis or application-specific knowledge, it is unlikely to be able to do anything other than simple optimization, perhaps similar to those defined in Linux (see Section 2.1). Furthermore, optimizations usually performed for single processor systems may not hold in NUMA architectures.

The second is to try to integrate Ada's notion of partitions into the notion of regions of coherence. An active partition currently represents a single address space. Given its SMP roots, we could require that this maps to a *logical region of coherence*. In that, the language/compiler must define and ensure the occasions when tasks sharing memory must see a consistent view. This then must be provided by the underlying architecture directly, or enforced by the compiler (perhaps though memory fences). From the programmer's view point, all main memory access are assumed to have similar access times, coherent caches allow for optimizations and there is no significant penalty from migrating a thread from one processor to another.

However, even within a cc-NUMA it may be that we want to separate this single address space so that we know which processors are local to which addresses. Hence we could consider dividing an active partition up into one of more *regions* or *locales.* Within a region all memory access is guaranteed to be uniform. Memory can be shared between regions but cross region accesses would be non uniform. Although, it is tempting to pursue this approach, the changes required to the language are likely to be significant.

The third approach is to work with the facilities already provided by Ada to control the locations of tasks and data. This is the approach adopted by the paper. Several issues are identified, namely:

- Understanding the address map.
- Using storage pools to partition the heap.
- Using representation aspects to control object location.
- Using task affinities to control thread migration.

Each of these items is addressed in the following section.

4.1 Understanding the address map

In a cc-NUMA architecture, a memory address will be segmented in some way so that the memory banks can be identified. In our example architecture, each node in the system has an attached processors (with two cores) and a memory bank. The high-order bits of the physical address allows the node to be identified. Ada allows an implementation to reflect this in its definition of the `Address` type in package `System`.

```
-- In package System
type Address is implementation-defined;
Null_Address : constant Address;

-- In package System.Storage_Elements
type Storage_Offset is range
     implementation-defined;
type Integer_Address is implementation-defined;
function To_Address(Value : Integer_Address)
         return Address;
function To_Integer(Value : Address)
return Integer_Address;
```

The type `Address` represents machine addresses capable of addressing individual storage elements. `To_Address` is intended for use in address clauses. Implementations can overload `To_Address` if appropriate. For example, the Annotated Ada Reference Manual indicates that on a segmented architecture, it might make sense to have a record type representing a segment/offset pair, and have a `To_Address` conversion that converts from that record type to type `Address`. However, it also indicates that “the validity of a given address depends on the run-time model; thus, in order to use `Address` clauses correctly, one needs intimate knowledge of the run-time model.”

From our perspective, what is important is to be able to determine the distance of a processor from a memory bank servicing a particular address. Here, for simplicity, we assume that `Distance` and `Memory_Bank_Id` are predefine type.

```
type Distance is
     range 0 .. implementation-defined;
type Memory_Bank_Id is
     range 0 .. implementation-defined;
```

And that each processor has an identifier:

```
type Processor_Id is
     range 0 .. implementation-defined;
```

Now we assume the following subprograms:

```
function Get_Distance(From_Proc : Processor_Id;
  To_Memory : Memory_Bank_Id) return Distance;
function Get_Memory_Bank(Addr : Address)
  return Memory_Bank_Id;
function To_Address(Memory : Memory_Bank_Id;
  O : Storage_Offset) return Address;
```

A distance of 0, implies that the memory is local to a particular processor. A distance of one implies, that the memory is one-hop away and therefore the access time is slower. A distance two is two hops away. Hence, in the event of a cache miss, the Memory Access Time for Distance 0 $<$ Memory Access Time for Distance 1 $<$ Memory Access Time for Distance 2 .. $<$ Memory Access Time for Distance n.

4.2 Using storage pools to partition the heap

In a cc-NUMA-aware program, it is necessary to have control over where objects are allocated on the heap. Currently, Ada allows an application to partition the heap and to use its own storage management algorithms. However, it does not go as far as allowing an application to specify the location of the storage pools. [1]

First we briefly review the current storage pools, and then we suggest a simple extension.

The interface to storage pools is given below

```
with Ada.Finalization;
with System.Storage_Elements;
package System.Storage_Pools is
  pragma Preelaborate(System.Storage_Pools);
  type Root_Storage_Pool is abstract new
      Ada.Finalization.Limited_Controlled
      with private;
  pragma Preelaborable_Initialization(
         Root_Storage_Pool);

  procedure Allocate(
     Pool : in out Root_Storage_Pool;
     Storage_Address : out Address;
     Size_In_Storage_Elements : in
          Storage_Elements.Storage_Count;
     Alignment : in
          Storage_Elements.Storage_Count)
          is abstract;

  procedure Deallocate(
     Pool : in out Root_Storage_Pool;
     Storage_Address : in Address;
     Size_In_Storage_Elements : in
          Storage_Elements.Storage_Count;
     Alignment : in
          Storage_Elements.Storage_Count)
          is abstract;

  function Storage_Size(Pool : Root_Storage_Pool)
     return Storage_Elements.Storage_Count
     is abstract;
private
... -- not specified by the language
end System.Storage_Pools;
```

A programmer can define their own allocation and deallocation algorithm by extending the `Root_Storage_Pool` type and providing concrete implementation for the `Allocate` and `Deallocate` subprogram.

The Ada Reference manual defines for every access-to-object subtype S, the following representation attributes

- `S'Storage_Pool` – Denotes the storage pool of the type of S. The type of this attribute is `Root_Storage_Pool'Class`.

- `S'Storage_Size` – Yields the result of calling `Storage_Size(S'Storage_Pool)`, which is intended to be a measure of the number of storage elements reserved for the pool. The type of this attribute is universal integer.

[1]Of course, if the user is statically defining an array of bytes then an address attribute definition clause can be used.

There are several points worth making

1. More than one access type can be allocated to the same pool.
2. If a storage pool is not specified for an access type, then the implementation chooses a standard storage pool for it in an implementation-defined manner. *It is not possible to specify the address of a storage pool.*
3. There is no default implementations for the `Allocate` and `Deallocate` subprograms.

Given a programmer's understanding of the address map (see section 4.1) it would be possible to control partitioning of the heap by allowing an attribute to define the address of a storage pool. Furthermore, if the default implementation techniques were to made available by providing concrete subprograms (in `System.Storage_Pools`), then again a programmer could specify the location of the pool without have to provide an allocation and deallocation algorithm. However, we again stress that this can only be safely done if the programmer is aware of the run-time model.

4.3 Using attribute definition clauses to control object location

Representation aspects in Ada are a compromise between abstract and concrete structures. Three main representational aspects are available:

(1) Attribute definition clause

(2) Enumeration representation clause

(3) Record representation clause

Here, we are concerned with attribute definition clauses as they allows various attributes of an object, task or subprogram to be set; in particular the storage alignment, the maximum storage space for a task, and the address of an object. If an implementation cannot obey a specification request then the compiler must either reject the program or raise an exception at run-time.

Given the programmers understanding of the address map, it is therefore relatively straight forward to control where object and code is placed.

4.4 Using task affinities to control thread migration

Using tasks affinities it is easy to control the migration of a task. This issue will be not discussed in this paper. See [14] for a discussion on mapping issues to current platforms and [5] for a discussion on appropriate facilities to support schedulability analysis.

5 General Memory Architectures

Once we go beyond cc-NUMA architectures towards heterogenous processor and memory, the issues become more complex and the associated schedulability analysis and worst-case execution time analysis more difficult.

The key issues are:

- Does the underlying platform support a single address space model? If it does *not*, then the program can consists of multiple Ada partitions, and a distributed systems model seems appropriate.
- If the underlying platform *does* support a single address space model, does the underlying platform provide a *single* physical region of coherence? If it does, then the facilities discussed so far in this paper would seem appropriate, but only if the programmer has detailed knowledge of the run-time models. If the program does not have detailed knowledge, then we believe that extensions to the Ada partition models may be appropriate to identify locales.

- If the underlying platform *does* support a single address space model and also supports *multiple* physical regions of coherence, then we believe that extensions to the Ada partition models may be appropriate to identify logical regions of coherence.

6 Conclusions

In recent years, Ada's popularity has waned. However, as architectures become more parallel and heterogeneous, the language is almost unique in that its semantics are already defined for SMP systems. Ada also provides a comprehensive sent of facilities for mapping the programmers' abstract data types onto the concrete implementations of a given platform.

Unfortunately, it can be argued that Ada doesn't provide enough expressive power to cope with NUMA architectures: both current cc-NUMA systems and future, potentially, heterogenous general NUMA architectures. This paper has undertaken an initially exploration of some of the issue, with the goal of initiating discussion at the workshop. It is clear that the level of program control that we have specified here is only really useful if the programmer has detailed knowledge of a compiler's run-time model, and the compiler and run-time has been written to facilitate such interactions.

References

[1] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification, Version 0.618, April 2005. `http://research.sun.com/projects/plrg/fortress0618.pdf`.

[2] AMD. BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors, May 2003. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26094.PDF`.

[3] AMD. Performance Guidlines for AMD Athlon 64 and AMD Opteron ccNUMA Multiprocessor System, 2006. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40555.pdf`.

[4] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[5] A. Burns and A.J. Wellings. Supporting execution on multiprocessor platforms. In *IRTAW 14*, 2009.

[6] Pat Conway and Bill Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2):10–21, 2007.

[7] Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. Advanced Configuration and Power Interface Specification, Revision 2.0, July 2000. `http://www.acpi.info/DOWNLOADS/ACPIspec30b.pdf`.

[8] R.E. Diaconescu and H.P. Zima. An approach to data distributions in chapel. *Int. J. High Perform. Comput. Appl.*, 21(3):313–335, 2007.

[9] Hewlett-Packard. HP ProLiant DL585 Server Technology Technology Brief, February 2006. `http://h20000.www2.hp.com/bc/docs/support/SupportManual/c00180597/c00180597.pdf`.

[10] IBM. Report on the Experimental Language X10, draft v 0.41, Feb 2006. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html/\$FILE/ATTH4YZ5.pdf`.

[11] Andi Kleen. An NUMA API for Linux, August 2004. `http://www.halobates.de/numaapi3.pdf`.

[12] Christoph Lameter. Local and Remote Memory: Memory in a Linux/NUMA System, June 2006. `http://www.kernel.org/pub/linux/kernel/people/christoph/pmig/numamemory.pdf`.

[13] J. Real and S. Mitchell. Beyond ada 2005 session report. In *Proceedings of IRTAW 13, Ada Letters, XXVII(2)*, pages 124–126, 2007.

[14] A.J. Wellings and A. Burns. Beyond ada 2005: allocating tasks to processors in smp systems. In *Proceedings of IRTAW 13, Ada Letters, XXVII(2)*, pages 75–81, 2007.

[15] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. The SGI AltixTM3000 Global Shared-Memory Architecture, 2003. `http://care.sgi.nl/pdfs/3474.pdf`.