Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 51b

Systems Group, Department of Computer Science, ETH Zurich

Automated Performance Discovery For a Multicore OS

by

Robert Meier

Supervised by

Adrian Schüpbach, Timothy Roscoe

Spring Semester 2012

Informatik
Computer Science

Contents

1 Introduction

For a couple of years multicore systems have been the de facto standard and today even in small personal computers the number of cores grows at quick pace [3]. Researchers are working with chips that have more than 48 cores [13,15] and the limit certainly has not been reached yet. One among many reasons for this development is that several, relatively slow processors can be cooled more efficiently than a single, very fast one. But it is a well known fact of course that working with more than one core can be quiet tricky. Intuitively one would think for example that two processors, in comparison to a single one, could do twice as much work in the same amount of time, but this has long ago been proven to be a fallacy [1].
While application programmers have a hard time conquering parallelism, operating systems (OS) are presented with an at least equally difficult task of managing all available resources of a given system. It has been pointed out [3,5] that hardware will be no longer homogeneous in the future. Heterogeneous cores on the same chip could be used to perform differing tasks or some cores could even be switched off entirely to reduce power consumption. This means that the cores have potentially different instruction sets as well, which poses an additional problem for OSs. An intuitive example would be a Graphics Processing Unit (GPU), that was originally designed for a very specific task.
Barrelfish is a research operating system released by ETH Zurich that explores new architectural ideas to cope with modern hardware. It introduces the notion of a Multikernel which (simplified) is the idea of running a lightweight kernel on every single core. The kernels run independently from one another, i.e. there is no shared state between them and a consistent view is achieved with message passing. Together all these different mini-kernels form the OS, which can now be viewed as a distributed system. Please refer to 3.1.1 and [4] for a more detailed explanation.
The purpose of this thesis was to implement a so-called datagatherer for the Barrelfish OS. The System Knowledge Base (SKB) is a program that stores data about the hardware the OS is currently running on. For example it could store the size of the data caches of a certain processor and make this information available to other programs. Based on this knowledge it is possible to adapt their behaviour according to it. A datagatherer is a program that queries the hardware for all the desired information and stores it in the SKB.
Every bit of information is potentially important, in other words there might exist an application that can benefit from it. An example are the cache parameters mentioned earlier. However there is of course a lot more that can be stored. An other fact that can be used extensively is which processors have shared caches. If two threads use the same data it is obviously beneficial if they run on two processors that share one or more caches.
To make it possible for applications to apply further optimizations cache access times are also collected by the datagatherers. This is done by a series of tests to measure the latencies of all cache levels.
For this thesis we implemented the part of the datagatherer that gathers in-

formation about Intel hardware only. Different hardware represents the same information differently, for example Intel uses integer codes to describe the cache parameters. To avoid potential problems all the data is stored in an abstract format (3.4.2), that is independent from hardware vendors. The datagatherers transform the data into this generic format, but they do not interpret it at all. For example cache identifiers are stored but shared caches are not computed.
Predefined queries, expressed in the logical programming language Prolog, are used to retrieve the data. Prolog makes it possible for very complex statements to be expressed in a relatively simple and intuitive way. That makes this language perfect to retrieve and interpret data at the same time.

The remainder of this document is structured in the following way:
The Background section (3) gives an introduction to the concepts that are relevant for this document. Not everything can be covered thus the Related Work section (4) provides pointers to several other sources for further explanation.
The Design (5) and Implementation (6) sections describe in detail how the datagatherer was built. After evaluating (7) the data, we draw the conclusion (8) and outline future work (8.1).

2 Motivation

The purpose of this thesis was to implement a datagatherer for the Barrelfish operating system. A datagatherer is a specialized program designed to detect various features about the hardware the OS is currently running on. The System Knowledge Base (SKB) stores this data in a machine independent format (3.4.2) in order to make it available to other parts of the OS and in general to every kind of application. Sophisticated queries, expressed in the logical programming language Prolog, are used to access and interpret the data.

There are several good ways to optimise a program. An appropriate algorithm combined with adequate datastructures is a good basis to start optimising a program. The hardware that executes an application has many factors that have to be taken into account when trying to get optimal performance. To put it simple: "constants matter". This means that at some point it becomes necessary to take a closer look at the details of the hardware that executes a program.
A specific example of important hardware parameters that could be of interest would be the details of the cache. Programs pay a very high performance penalty if the data they access is not present in the caches, but in main memory, for example. In other words it is beneficial for a process to have the data it is working on as high up in the memory hierarchy as possible.
A classic example is the traversal of a matrix. In C, matrices are stored in row-major order. This means that when traversing the matrix (of byte entries) row after row, we access the memory in sequential order. This is favourable, since we access the data along the cache lines. Assuming we have a line size of 64bytes we have one cache miss every 64 accesses. If we do the traversal column wise however we are no longer traversing the memory in sequential order. That means that if the matrix is big enough we have one cache miss in every access.
The above example illustrates the point that algorithms should be implemented in a cache aware way. Or more general that they have to take hardware details into account.
The problem with this kind of information (cache parameters, etc.) is that it is inherently hardware specific. If such optimisations are applied to a program it gets tied to a specific machine. Very often though, one needs programs that are portable across all kinds of hardware. We believe that there is a way to make such low level optimisations possible while still keeping programs portable.
Our approach is as follows: A database, that runs in the background, stores information about the hardware the OS is currently running on. It would for example store the size of all the caches. An application can query this database and adapt itself according to this information. The database stores the information in an abstract format, meaning that it does not contain hardware specific details. Applications can use this data to apply optimisations specific to the hardware while at the same time staying at a level of abstraction that ignores how the underlying system actually looks like. The optimal result would be an application that is portable and runs efficiently on every type of platform.

The Barrelfish OS, in combination with the SKB, provides exactly this interface. Datagatherers that are specially tailored to the hardware are executed and fill the SKB with as much useful information as possible. All the hardware specific details are encapsulated within these programs, thus allowing the rest of the applications to run at a higher level of abstraction.

3 Background

3.1 Barrelfish

Barrelfish is a research operating system released by ETH Zurich. It was built to investigate new architectural ideas for future multi- and many-core systems. While classical operating systems like Linux, etc. tend to be rather monolithic, Barrelfish embraces the idea of being a distributed system. There are several reasons to abandon the classical view of operating systems as static constructs. The main one being the fact that modern hardware looks more and more like a networked system [5]:

3.1.1 The Operating System As a Distributed System [5]

Historically a distributed system is characterized by three key features: **heterogeneous** nodes, communication **latency** between them and the **dynamic change** of the number of nodes.
It has been pointed out [3, 5], that future hardware will no longer consist of homogeneous chips but rather of different components, each of which built for a specialized task. An intuitive example for this is a GPU. The problem with heterogeneous CPUs for an OS is that the instruction sets are potentially different. Therefore the same kernel can in general not be executed on every chip. Barrelfish introduces the notion of a cpudriver which is essentially a very lightweight kernel that is executed on every core. As the term suggests, a CPU is thought of as just another device that has to be handled by the OS and hence needs a driver that manages it. Heterogeneity is no longer a problem since the driver can (must), according to its definition, handle the hardware it runs on.

On modern NUMA machines CPUs are grouped into so called NUMA - Domains each of which is physically built around a part of main memory (Figure

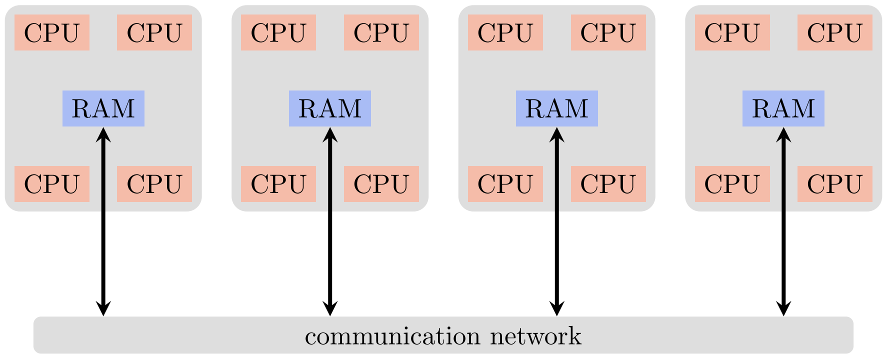

Figure 1: Numa Domains

1). Therefore processors access the part of main memory that is in their NUMA - Domain faster than memory in other domains. These different access times can be viewed as latencies that are present in the system.
As the number of cores increases, the probability that one of them fails increases as well and hence we suddenly have hardware that might “loose” a CPU in the middle of execution. The big difference to traditional hardware is that a crash of the CPU is (should be) no longer equivalent to a crash of the whole system. It is entirely possible and of course very desirable that the system can recover and continue to work after it has lost one or more processors (nodes in the distributed system).
We see that modern hardware exhibits important features of any distributed system and therefore has to be viewed as a distributed system, as well.

3.2 System Knowledge Base (SKB)

The System Knowledge Base (SKB) [14] is a novel feature of the Barrelfish operating system. It is essentially a database that stores information about the hardware the OS is currently running on. It uses specialized programs (data-gatherers) to collect relevant information which is stored as Prolog Facts (3.3.1). Sophisticated Prolog - queries can then be used to retrieve it. To interpret the Prolog code, the SKB uses Eclipse CLP [2, 7].
Prolog is a logic programming language that allows for very complex queries to be expressed in a fairly simple and intuitive way. Please check section 3.3 for a short introduction to this language.
A simple example of how the SKB could be used: An application needs to run in parallel on two cores. It is known in advance that both cores will often access the same data. Therefore it would be useful to have a shared cache between them. In this example the application would first send a query to the SKB, that will then return a pair of cores that satisfy the requirement of sharing one or more caches. Using this information the application can choose the appropriate cores.
The SKB makes it possible to adapt the behaviour of an application directly to the underlying hardware. Note that the logic which reacts to information from the SKB (i.e. the part of the application that changes the behaviour of the program) can be programmed in a very generic way, since information stored in the SKB is kept in a format that is independent (3.4.2) of the underlying hardware.

3.3 Prolog

This section gives a very brief introduction to the Prolog programming language. For a complete and more detailed description, please refer to other documents. A little tutorial can be on the website “Learn Prolog Now!” [6].

3.3.1 Prolog Facts

A Prolog program consists of a **Knowledge base** and a number of queries. The knowledge base constraints a set of **facts**. For example a simple knowledge base, consisting of 2 facts:

```
colour(red).
colour(magenta).
```

A fact is assumed to be true. We now have an exemplary knowledge base that stores the fact that red and magenta are both colours.
Queries are used to "ask questions" about the state of the knowledge base. To answer the questions, the interpreter looks at the available facts and tries to satisfy the query. For example, when asking :

```
?- colour(black).
```

the response will be

```
no.
```

since there simply is no fact stating that black is a colour. But of course when asking if magenta is a colour, the query can easily be satisfied:

```
?- colour(magenta).
yes.
```

It is also possible to search for facts, that satisfy a given rule:

```
?- colour(What).
What = red ? ;
What = magenta
```

The results are red and magenta, as both would satisfy the query: "colour(red)" and "colour(magenta)" are both facts that are present in the knowledge base.
This very powerful feature makes it possible to interpret facts according to a given constraint. We will see that this is precisely what is needed in the context of the SKB.
Let us consider a more complex example where we store information in the following form:

```
cache(core_nr, cache_level, cache_size, cache_id).
```

Think of a system with 4 cores, each of which has 2 caches. We create a little knowledge base:

```
cache(1, 1, 8192, 1).
cache(2, 1, 8192, 2).
cache(3, 1, 8192, 3).
cache(4, 1, 8192, 4).
cache(1, 2, 524288, 5).
cache(2, 2, 524288, 5).
cache(3, 2, 524288, 6).
cache(4, 2, 524288, 6).
```

Note the shared caches between nodes 1 and 2 and between 3 and 4, respectively. We define a rule to check whether two cores share a cache:

```
sharing_cache(CoreA, CoreB) :-
    cache(CoreA, _, _, ID),
    cache(CoreB, _, _, ID).
```

The interpreter tries to satisfy it with the facts stored in the knowledge base (the "_" is a "don't - care"):

```
sharing_cache(1, 2).
yes.
```

There are two facts that satisfy the rule, namely:

```
cache(1, 2, 524288, 5).
cache(2, 2, 524288, 5).
```

If we have one core we could also use the query to find another core that shares a cache with it:

```
sharing_cache(3, Other).
Other = 3 ? ;
Other = 3 ? ;
Other = 4
```

The query is of course a bit too simple to do the job properly: we already know that core 3 shares two caches with itself. Note that core 3 has 2 caches and therefore the answer "3" is returned two times. Apart from this little extra information we get the desired result and now know that core 3 shares a cache with core 4.
This very powerful feature is exactly what we need for the SKB in order to provide complex queries. Please refer to section 6 for more details about the queries that are used to retrieve information.

3.4 Datagatherer

The purpose of this thesis was to design (5.1) and implement a program to gather data about the hardware and (3.4.2) to decide how the data should be stored and accessed.
The programs were written for Intel hardware only. Code to analyze other hardware is present, but was not part of this thesis.

3.4.1 Gathering Data

To collect as much data as possible, the SKB runs specialized programs called datagatherers on every core. These programs run independently from one another and write all data directly to the SKB . The data is not interpreted at all, it is simply transformed into a generic format (3.4.2).
A central instrument to get information about the hardware is the CPUID instruction (CPUID instruction). It provides a very simple way to collect a lot of information, be sure to check section 3.5 for a more detailed description.
To measure cache and memory latencies we implemented algorithms that adapt to the underlying hardware, check section 6.4 for more details.

3.4.2 Storing Data

Data is collected by the datagatherers and then stored in the SKB. It is important that the data is stored in a way that is independent from the underlying hardware.
An intuitive example is information about the processor cache. When using the CPUID instruction on an Intel processor to gather cache information (with CPUID(2)), the result consists of hexadecimal codes that refer to a specific cache configuration. The value 0xd, for example, refers to an L1 data cache whose size is 16KB, 4-way set associative and has a line size of 64 bytes [10].
In a first attempt one could simply store the value 0xd in the SKB:

$$\text{cache}(0\text{xd}).$$

but this would bring about several problems. First of all the queries that access the data became unnecessarily complex, since the corresponding configuration would have to be determined. The even bigger problem however, is that the value 0xd probably only makes sense for an Intel processor. When looking at a broader context (3.1.1), where there might be different cores with differing instruction sets in the same system, it becomes necessary to deal with all special cases at the "query level", e.g. one would need a different query for every kind of hardware.
The previous approach is clearly a bad idea. Instead we store the information in a format that is a bit more abstract. In the above example this could be done in the following way:

$$\text{cache}(1, \text{“data”}, 16384, 4, \text{“s”}, 64).$$

All information is available and the format is sufficiently generic to write abstract queries.
Note that in this setting the interpretation of the code is done by the datagatherer. It makes sense to perform this work at this level because the datagatherer is specialized to the hardware anyway, whereas the SKB should be at a higher level of abstraction. In other words the SKB remains independent from hardware. This transformation is the only interpretation that is made by the datagatherers. Every other processing of the data is done by higher level code,

predominantly written in Prolog.

3.5 The CPUID Instruction

This section gives an overview of the CPUID instruction. Some examples are taken directly from Intel Documentation [10].

The CPUID instruction is used to gather information about the processor the program is currently running on. The *eax* register determines the function that is executed. When it is initialized to 0, written CPUID(0), for example, the instruction returns information about the vendor. It would be executed like this:

```
movl $0, %eax
cpuid
```

The result is written into the registers *eax*, *ebx*, *ecx*, *edx*. In the above example, after the instruction has been executed the registers would look like this:

ebx	0x756E6547
ecx	0x49656E69
edx	0x6C65746E

When we interpret the integers as characters we get:

ebx	u (75)	n (6E)	e (65)	G (47)
ecx	I (49)	e (65)	n (6E)	i (69)
edx	l (6C)	e (65)	t (74)	n (6E)

Which corresponds to the ASCII string

GenuineIntel

Some functions need an additional argument that is passed via the *ecx* register. Later we will use this second parameter extensively, but for now it is not very important.

The CPUID instruction provides access to a wide variety of information, section 6 will cover the ones that were used in the context of this thesis.

3.6 Normal Forms

In this section we give a very brief overview of normal forms. We neither can nor do we want to give a complete introduction to the topic in this document. This section is merely intended to be a reminder that summarizes some important points. A very good and detailed explanation can for example be found in [11]. There are of course numerous other documents that provide an equally well structured introduction to the topic.

The examples that follow are inspired by [11].

Relations in a (relational) database can be re-organized (**normalized**) to reduce redundancies and functional dependencies. Depending on which properties are satisfied, the schema is said to be in a specific normal form:

First Normal Form [11]

A relation is in first normal form (1NF) if and only if every attribute consists of atomic values. The following relation

Family		
Father	Mother	Child
Felix	Joana	{Robert, Marc}
Charly	Karin	{Nils, Jens}

is not in 1NF since the "child" attribute is composed of two values. Yet we can easily transform the relation to meet the requirements of the 1NF:

Family		
Father	Mother	Child
Felix	Joana	Robert
Felix	Joana	Marc
Charly	Karin	Nils
Charly	Karin	Jens

Second Normal Form [11]

Intuitively a relation is in the 2NF if it is in the 1NF and if it contains information about only one concept. This means that there exists no non-prime attribute that is dependent on a proper subset of any candidate key. For example this relation:

Courses			
StudentNr.	CourseNr.	Student Name	Semester
181	5	Nina	8
151	1	Robert	6
151	2	Robert	6
151	3	Robert	6
131	2	Stefan	6
131	4	Stefan	6

is not in 2NF since there are the following functional dependencies:

$$\{StudentNr\} \rightarrow \{StudentName\} \text{ and } \{StudentNr\} \rightarrow \{Semester\}$$

Consequences: The problem with the above relation is that there can be update anomalies. If we would like to add the last names of the students, for example, we would have to do this in several different places. To change "Robert" to "Robert Meier", for example, we would need to update three different fields in the table.

Third Normal Form [11]

A relational schema $\Re$ is in 3NF if for all functional dependencies of the form $\alpha \rightarrow B$, where

$$\alpha \subseteq \Re$$
$$B \in \Re,$$

one of the three conditions holds:

1. $B \in \alpha$ (the functional dependency is trivial)
2. B is prime
3. α is a superkey of $\Re$

This means that $\Re$ has to be in 2NF and every non-prime attribute has to be directly dependent on every superkey of $\Re$. Consequently no non-prime attribute provides a fact about a set of attributes that is not a key (Or: “Every non-key attribute must provide a fact about the key, the whole key and nothing but the key” [12]).

Boyce - Codd Normal Form [11]

To be in the Boyce-Codd normal form, a schema $\Re$ needs to satisfy even stronger conditions. Facts are then stored **exactly once** in the database. For every functional dependency $\alpha \rightarrow \beta$ at least one of the two conditions must hold:

1. $\beta \subseteq \alpha$ (the dependency is trivial)
2. α is a superkey of $\Re$

4 Related Work

Operating System As a Distributed System

Baumann et. al. [5] introduced the idea of viewing a modern OS as a distributed system. The main reason for this is that hardware looks more and more like a networked system. Different types of cores (e.g. GPUs, FPGAs), that may be plugged into the system **dynamically** introduce **heterogeneity**. Moreover cache-coherent NUMA machines lead to different memory access times which is essentially the same as **communication latency**.
Thus modern hardware exhibits key features of a distributed system and the OS on top of it should therefore resemble a distributed application.
Another example of a distributed OS is the idea of "Factored Operating Systems" (fos) [16]. To achieve scalability they use the idea of distributed internet services. A so-called server runs on every core which provedes the kernel services and together they form the OS.

System Knowledge Base

A special feature of the Barrelfish OS is the System Knowledge Base [14]. Specialized programs analyze the hardware and gather data that is stored into this database. Sophisticated queries allow applications to access the information at runtime and adapt their behaviour according to the underlying hardware.

Cache Aware Algorithms

There is a great number of cache aware algorithms and systems. [8,17,18] some are examples of such systems. Many of the algorithms have to be compiled for every system they run on to optimally use the properties of the hardware. The SKB goes one step further and provides desired parameters at runtime. Applications that use the SKB no longer have to be custom tailored to every system but can conveniently adapt themselves at runtime.

Measuring Latencies

Yotov et. al. [19] describe a set of algorithms to measure cache and memory latencies. The paper describes in detail how latency measurements of all cache levels can be obtained. For this thesis we adapted (5.2) the algorithms a little bit since we do not need their complete functionality. The original algorithm derives system parameters, in this case however this is unnecessary because they are already stored in the SKB by the time the algorithm is executed.

5 Design

This section covers the design of the datagatherer. Details about the implementation are covered in section 6.
As mentioned before, we focus on Intel hardware only.
The datagatherer is a relatively simple program that instantiates itself on every core and gathers information. The instances run mostly independent from other instances and save the information directly to the SKB.
Conceptually, the datagatherer consists of two parts: The first part (5.1) simply queries the hardware and stores the information. The second component (5.2) performs measurements (cache latencies, etc.). It makes extensive use of information gathered earlier by the first part of the program. This part is executed sequentially. That means that no two instances of the datagatherer can execute it simultaneously. We need this because we have to avoid bus contention that would yield very inaccurate measurements.

5.1 Gathering Data

The first part of the datagatherer simply queries the hardware and stores the information directly to the SKB. Here we frequently use the CPUID instruction. This function is called with one or two parameters. The first one, passed in *eax*, determines the method that is executed, i.e. what kind of query is executed, while the second one provides additional information for some queries. For example we can execute the instruction with first parameter 4 to determine some cache parameters. The second argument then further specifies the cache that we want the instruction to be executed upon.
Not every processor provides the full functionality of the CPUID instruction. Which functions are supported can be determined by querying CPUID directly. The result tells us which integers can be passed in the *eax* register as the first parameter. The datagatherer checks the availability of the methods and only executes those that are present.
We created a wrapper function for every CPUID - method to make it easy to selectively execute single methods.
Cache parameters are gathered at the end of this stage. To be able to do it properly we need to perform some additional work, that relies on previously gathered information.

5.2 Measuring Latencies

The second part of the datagatherer performs latency measurements with the algorithms described by *Yotov et al.* [19]. For this we heavily rely on previously gathered data, for example the algorithms need the capacity and the associativity of all caches.
Note that this task cannot be done in parallel on all cores at the same time because we have to avoid bus contention to get accurate measurements.
To measure the access times of the caches we obviously need a function that

can measure time. However this is a bit of a problem, since we do not actually know which methods are available on the current processor.
At this point we know that the datagatherer has executed the first part (5.1), thus the available features are stored in the SKB. The SKB itself provides a query that determines which function is available to measuring the time. We call this a **measuring strategy** and a simple integer code is used to represent a specific strategy:

```
measurement_strategy(StrategyCLOCK, StrategyRDTSC,
                     StrategyRDTSCP, Apic, Strategy) :-

    feature(Apic, "rdtscp") ->
            Strategy is StrategyRDTSCP;
    feature(Apic, "tsc") ->
            Strategy is StrategyRDTSC;
    Strategy is StrategyCLOCK.
```

This query checks which features are available and then tells us the best strategy. In this context “best” means most accurate. The integer constants representing the strategies are not fixed. That is why the query takes them as arguments. We think it is better to avoid setting the constants, since the program that executes the query is more flexible that way.
With this version of the query we have three alternatives to measure time: the clock, rdtsc and rdtscp instruction, respectively. More could easily be added. The problem with this approach is that the query has to be adapted as well. This is tricky since the validity of the query-call, in the C code, for example, is not checked at compile time.

5.2.1 Overhead

An inevitable problem with measuring the time are inaccuracies. We try to keep them at a minimum, nevertheless our measurements suffer mostly from either method-call overhead or loop-overhead.
To measure cache levels lower than L1 we have the problem that we need to access the data in special ways to avoid cache hits in higher cache levels. To access the L2 cache, for example, we need an L1 miss. To achieve this we use (6.4) the algorithms of *Yotov et al.* [19]. These algorithms need information about the hardware (cache sizes, associativity, etc.). The problem is that we do not know these values at compile time and thus cannot use a macro to forego using a loop, which incurs (loop-) overhead.
The other source of inaccurate measurements is the measuring itself. In other words: the starting and stopping of the clock has a certain amount of overhead. One way to minimize the effect of this overhead is to execute n instructions and then divide the measured time by n. Let T be the actual execution time of an operation, A the overhead of taking the time and n the number of executions. Then the measured execution time M of one instruction is:

$$M = \frac{nT + A}{n} \Leftrightarrow$$

$$M = T + \frac{A}{n}$$

So we see, that if n is big, the effect of A on a single instruction becomes small. Loop overhead on the other hand is independent from the number of iterations since we are dealing with compare, add and jump instructions which occur in every iteration. However the operations on the control variables of a loop are mostly independent of the instructions in the loop body and therefore can be executed in parallel, or out-of-order (whatever the processor chooses to do). Thus it is not quiet clear how much overhead is actually caused by the loops, especially when considering speculative execution, for example.
At the beginning we measure both the loop and the timing overhead and store the values in the SKB. When taking measurements we can adapt the values a bit to make them more accurate (we subtract the overhead from the measured time).

6 Implementation

As mentioned already before this thesis focuses on Intel hardware. We first describe how we collect general information about the hardware with the CPUID instruction. We then show how shared caches can be determined and at the end of this section it is demonstrated how the access times to caches and to main memory are measured. Throughout this section we present the Prolog queries that are available to retrieve the data that is collected.

6.1 General Information

Right at the beginning we can do the two following things: first we can save the vendor information. For that we store this simple fact to the SKB:

vendor(ApicId, "intel").

and provide the following query to access the information:

```
get_manufacturer(Apic, Manufacturer) :-
            vendor(Apic, Manufacturer).
```

Secondly, we determine some facts about the current cpu thread that executes the datagatherer. We store the information in the following form:

cpu_thread(ApicId, PackageId, CoreId, ThreadId).

To find the package of a specific core we provide the following query:

```
get_package_of_core(Apic, Package) :-
            cpu_thread(Apic, Package, _, _).
```

6.2 Miscellaneous Information From CPUID

This section focuses on information that is gathered with the help of the CPUID instruction. Sometimes we omit little details about the use of the CPUID instruction if they do not play a central role for this application. For a complete description with all details, refer to the documentation [10].Very often one needs to check single bits to get information. In these cases we omit the details of the implementation since it would not be very interesting.

6.2.1 Determining Available Methods

The first thing that has to be done is to check which methods are applicable. For this we execute CPUID(0) and check the *eax* register for the "largest standard function number". Every method that is smaller (i.e. the first parameter of the CPUID instruction is smaller) than this number is supported by the processor. We use a simple switch statement to execute as much functions as possible:

```
//determine the highest function
uint32_t eax, ecx;
eax = 0;
ecx = 0;
cpuid(&eax, NULL, &ecx, NULL);
const uint32_t hightestFunction = eax;

//execute all functions that are available
switch(hightestFunction) {

        case 0xA: //add information about performance monitors
        case 9: //gather dca parameters
        case 8:
        case 7:
        case 6: //add information about thermal sensors and
                //power management capabilities
        case 5: //add information about monitor/mwait parameters
        case 4:
        case 3:
        case 2:
        case 1:  //add feature information
        case 0:
        default:
                break;
};
```

Note that we do not execute all possible methods. Some missing ones are executed in a different context (6.3).

6.2.2 Processor Features

Every processor has certain (special) features. For example it might support a specific instruction, like the rdtsc command that can help to measure cpu clock cycles.
To check which features are enabled or available on a specific logical core we can use the CPUID instruction with parameter 1. It returns two bitmaps in register *ecx* and *edx*. Every bit corresponds to a specific feature and if it is set, the feature is enabled.
In our code we first define two string arrays to store the features:

```
    const char* featuresECX[] = {"sse3", "pclmuldq", "dtes64",
    "monitor_mwait", "ds_cpl", "vmx", "smx", "eist", "tm2", "ssse3",
    "cnxt_id", 0, "fma", "cx16", "xtpr", "pdcm", 0, "pcid", "dca",
    "sse4_1","sse4_2", "x2apic", "movbe", "popcnt", "tsc_deadline",
    "aes", "xsave","osxsave","avx", 0,  0, 0};

    const char* featuresEDX[] = {"fpu_x87", "vme", "de", "pse",
    "tsc", "msr", "pae", "mce", "cx8", "apic", 0,  "sep", "mtrr",
    "pge", "mca", "cmov", "pat", "pse36", "psn", "clfsh", 0,
    "ds", "acpi", "mmx", "fxsr", "sse", "sse2", "ss", "htt",
    "tm", 0, "pbe"};
```

We then traverse the registers and look at every bit. The bit offset is used as an index into the array to get the corresponding feature:

```
    for (int i = 0; i < total_nr_bits; i++) {

        //Check feature in ecx
        if((0x1&ecx) == 0x1 && featuresECX[i] != 0) {
            SKB_ADD_FEATURE(featuresECX[i])
         }
        ecx >>= 1;

        //Check feature in edx
        if((0x1&edx) == 0x1 && featuresEDX[i] != 0) {
            SKB_ADD_FEATURE(featuresEDX[i])
         }
        edx >>= 1;
    }
```

The facts that are stored into the SKB have the following form:

$$\text{feature}(\text{apicId}, \text{featureName}).$$

A concrete entry would then look like this:

$$\text{feature}(1, \text{"tsc"}).$$

Which would tell us that the rdtsc instruction is available on core 1.
To access the data we provide the following queries:

```
is_feature_available(Apic, Feature) :-
            feature(Apic, Feature).

get_all_features_of_apic(Apic, L) :-
            findall(feature(Feature), feature(Apic, Feature), L).
```

The first query is basically just a fact check. It returns “yes” if a given feature is available for a specific processor. To do the opposite, the second query returns a list of features of a specific processor (it actually “binds” L to the result).

6.2.3 MONITOR, MWAIT Parameters

When the CPUID instruction is executed with parameter 5 it will return information about the MONITOR / MWAIT instructions in the *eax*, *ebx*, *ecx* and *edx* registers.
The information is stored as simple facts of the form:

$$\text{monitor\_mwait}(\text{apicId}, \text{featureName}, \text{value}).\ \text{or}$$
$$\text{feature}(\text{apicId}, \text{featureName}).$$

Where featureName is the name of a parameter:

monitor_mwait(1, "smallest_monitor_size", 64).

The reason we create two different facts is because we like facts with a uniform format. Some features have a parameter, while others are simply present or not. For example processor 1 might support the MONITOR/MWAIT extensions:

feature(1, "monitor_mwait").

But to store the smallest monitor line size, for example we need an additional parameter:

monitor_mwait(apicId, "smallest_monitor_size", value).

The alternative would result in non-uniform facts

feature(1, "monitor_mwait").
feature(1, "smallest_monitor_size", value).

or in facts with potentially a lot of NULL values:

feature(1, "monitor_mwait", NULL).
feature(1, "smallest_monitor_size", value).

Therefore we need (want) two different facts.
To access the data we have a simple prolog query that allows to retrieve desired values:

```
get_monitor_mwait_value(Apic, Feature, Result) :-
        monitor_mwait(Apic, Feature, Result).
```

The information that is stored in the other format can be retrieved with the queries described in 6.2.2.

6.2.4 Digital Thermal Sensor and Power Management Parameters

The *eax*, *ebx*, *ecx* and *edx* registers return information about the digital thermal sensor and power management parameters when the CPUID instruction is executed with parameter 6.
This is again very "simple" data that can easily be stored in single facts of the form:

feature(apicId, featureName). or
thermal_power(apicId, featureName, value).

We use two different facts for the same reason as in 6.2.3.
To access the data we use the following query:

```
get_thermal_power_value(Apic, Feature, Result) :-
        thermal_power(Apic, Feature, Result).
```

Again, the information that is stored in the other format can be retrieved with the queries from 6.2.2.

6.2.5 Architectural Performance Monitor Features

To obtain information about performance monitor features, the CPUID instruction is executed with the parameter 0xa. The information is again very simple and we store it in facts of the form:

feature(apicId, featureName). or
performance_monitor(apicId, featureName, value).

Simple features are stored in the "feature" facts, while complex ones, having a parameter, are stored in a separate format. This is done for the same reasons described in 6.2.3.
We do not determine each version of the monitoring capability.
While we use this query

```
get_performance_monitor_value(Apic, Feature, Result) :-
        performance_monitor(Apic, Feature, Result).
```

to access the data, we use the queries from 6.2.2 to access the data that is stored in the other format.

6.2.6 Direct Cache Access (DCA)

DCA information can be gathered with CPUID(9). We store the information in the following form:

dca_parameter(apicId, featureName, value).

and use this query

```
get_dca_parameter_values(Apic, Feature, Result) :-
        dca_parameter(Apic, Feature, Result).
```

to retrieve the data.

6.2.7 Extended Features

The extended features are gathered with CPUID (0x80000001) and stored in the same way as all other features (6.2.2).

6.2.8 Advanced Power Management

At the time we wrote this document, the only information that was available from the CPUID(0x80000007) leaf, was whether "TSC Invariance" [10] was available or not.
In the code we perform a simple check on one bit to see if the feature is available or not. If it is, we add the following fact to the SKB:

feature(apicId, "invariant_tsc_support").

6.2.9 Address Sizes

The sizes of the physical addresses can be found out by executing the CPUID instruction with parameter 0x80000008. The *eax* register then returns the sizes and we store them in the following ways:

address_size(apicId, “physical”, value).

for the physical address size and

address_size(apicId, “virtual”, value).

for the virtual address size.
To access this data we define the two intuitive queries:

```
get_physical_address_size(Apic, Size) :-
        address_size(Apic, "pysical", Size).

get_virtual_address_size(Apic, Size) :-
        address_size(Apic, "virtual", Size).
```

6.3 Cache Information

This section covers the details relating to caches. We first describe how to get general data about the caches. Afterwards we explain how we can get the identifier of a specific cache.

6.3.1 Cache and TLB Descriptors

To best understand what a cache descriptor really is, we first look at a specific cache of the processor:

“1st-level data cache: 16-KB, 4-way set associative, 32-byte line size”

This ache descriptor tells us most of the things we need to know. The **level** of the cache, the **associativity** and the **line size**. Every cache in a system has such a descriptor.
To access this information we execute the CPUID instruction with parameter 2. CPUID returns the so called **“Cache and TLB Descriptor Decode Values”** in the *eax*, *ebx*, *ecx*, *edx* registers. These are simple 1-Byte integers that correspond to a cache descriptor: the value 0xC, for example, corresponds to the above descriptor. This means that there are potentially four descriptors per registers, although some might be marked as invalid. In our code we look at every register and extract the values.
Unfortunately there is no elegant way to map the descriptor decode values to descriptors, therefore we rely on a really big switch statement to do the job. This looks something like this:

```
switch(code) {

        /* ... */

        case 0xd:   return
                snprintf(destination, destinationLength, "cache(%d, 1"
                "\"data\", 16384, 4, \"s\", 64", apic_id);
        case 0xe:   return
                snprintf(destination, destinationLength, "cache(%d, 1"
                "\"data\", 24576, 6, \"s\", 64", apic_id);

        /* ...   */
};
```

Once the descriptors are ready for use, they are saved to the SKB in the following form:

$$\text{cache(apicId, Level, Type, Size, N, AssociativityType, LineSize).}$$

A specific example would look like this:

$$\text{cache(1, 1, "data", 16384, 4, "s", 64).}$$

This fact denotes a 16KB L1 data cache on core 1 (logical) that is 4-way set associative and has a line size of 64 Bytes.
To get this information from the SKB we provide the following Prolog queries:

```
get_nr_of_caches_of_type(Apic, Type, Nr) :-
    findall(Level,
    cache(Apic, Level, Type, _, _, _, _),L),
    length(L,Nr).

get_nr_of_caches(Apic, Nr) :-
    get_nr_of_caches_of_type(Apic, _, Nr).

get_capacity_of_biggest_cache_of_type(Apic, Type, Capacity) :-
    findall(C_Temp,
    get_capacity(Apic, _,Type,C_Temp), C_All),
    maximum_of_list(C_All, Capacity).
```

These are basic queries do determine the number of caches, or the capacity of the biggest cache. There are several special predefined queries to count specific cache types:

```
get_nr_of_instruction_caches(Apic, Nr) :-
    get_nr_of_caches_of_type(Apic, "instruction", Nr).

get_nr_of_non_instruction_caches(Apic, Nr) :-
            get_nr_of_data_caches(Apic, N1),
            get_nr_of_unified_caches(Apic, N2),
            (Nr is (N1+N2)).
```

Note: Some queries are omitted here because they only differ in one parameter: for example the "get_nr_of_data_caches" query would simply call the "get_nr_of_caches_of_type" query with a different parameter.
Several algorithms, e.g. the timing algorithm described in 6.4, need specific information about the caches, such as associativity and capacity. We provide some queries for that as well:

```
get_associativity(Apic, Level, Type, Assoc) :-
    cache(Apic, Level, Type, _, Assoc, _, _).

get_capacity(Apic, Level, Type, Capacity) :-
    cache(Apic, Level, Type, Capacity, _, _, _).

get_capacity_of_biggest_instruction_cache(Apic,Capacity) :-
    get_capacity_of_biggest_cache_of_type
        (Apic, "instruction", Capacity).
```

Some cache descriptors give some additional information about the cache. For example the value 0x25 tells us, aside from the previously mentioned information, that we have a sectored cache. To store this additional information we use a fact of the following form:

cache_detail(apicId, specialInformation).

As mentioned in 6.2.3, we use a new fact to avoid facts of variable format.
At the time we wrote this document there were two descriptors that contained information about the prefetching of the processor. If we encounter such descriptors, we store them in the following format:

prefetching(apicId, amount).

Where amount denotes how many **bytes** the processor prefetches.

6.3.2 Cache Identifiers

To get the cache identifiers we apply the same algorithm as Intel does in their "topology enumeration algorithm" [9]:
We iterate over all cache levels. In this context, the cache levels are called "subleafs". We use the CPUID(4, subleaf) function, where 4 is passed in the *eax* register and the subleaf, i.e. the cache level, is passed in the *ecx* register, to check if the current subleaf is valid. If it is we compute the cache id in the following way:

```
uint32_t eax, ebx, ecx, edx;

eax = 1;
ecx = 0;
cpuid(&eax, &ebx, &ecx, &edx);
const uint8_t initialAPICID = 0xff & (ebx >> 24);
```

```

eax = 4;
ecx = cacheLevel;
cpuid(&eax, &ebx, &ecx, &edx);

const uint8_t cacheType = 0xf & eax;

if (cacheType <= 0 || cacheType >= 4) {
        return 0; //the cache type is invalid
}

const char* cacheType[] = {
        "Invalid",
        "data",
        "instruction",
        "unified"
        };

const uint16_t cacheMaskWidth =
        log_roundToNearestPof2((eax >> 14) & 0xfff) + 1);
const uint32_t mask = ~((-1) << cacheMaskWidth);
const uint8_t threadsSharingCache =
        ((eax >> 14) & 0xfff) + 1;
const uint32_t cacheID = initialAPICID & (-1 ^ mask);
```

Note that we use a little helper function called "log_roundToNearestPof2". For a given argument n it first computes the smallest power-of-two integer a that is not smaller than n and returns $\log a$:

$$\text{log\_roundToNearestPof2}(n) = p$$

$$n \leq 2^p \wedge (\nexists q : n \leq 2^q < 2^p)$$

This information is then added to the SKB in the following way:

cache_identifier(apicId, cacheType, threadsPerCache, cacheId, level).

This is enough to define two queries that check whether two processors share a cache or not:

```
get_shared_cache(ApicA, ApicB) :-
        get_shared_cache_type(ApicA, ApicB, _).


get_shared_cache_type(ApicA, ApicB, Type) :-
        cache_identifier(ApicA, Type, ThreadsPerCache, Id, Level),
        cache_identifier(ApicB, Type, ThreadsPerCache, Id, Level),
        ThreadsPerCache >= 2,
        vendor(ApicA, Vendor),
        vendor(ApicB, Vendor).
```

We now have the cacheId but there is still a problem: In 6.3.1 we found out a lot about the caches in the system. We know for example if there is a L2 cache, etc. But we still do not actually have a mapping from cacheId to this information. The only thing we have is the cacheType and the cache level. We can combine this fact with one we stored before (6.3.1):

cache(apicId, "intel", Level, Type, size, N, AssociativityType, LineSize).

to get the desired mapping:

```
get_shared_cache_level(ApicA, ApicB, Level) :-
    get_shared_cache_type(ApicA, ApicB, Type),
    cache_identifier(ApicA, Type, Tp, Id, Level),
    cache_identifier(ApicB, Type, Tp, Id, Level).
```

In addition we provide more useful queries (the implementation not shown since they depend on many auxiliary functions):

```
get_how_many_shared_caches(ApicA, ApicB, Count) :-
        %how many caches are shared?

get_which_shared_caches(ApicA, ApicB, CacheLevels) :-
        %which levels are shared?

get_dont_share_caches(List) :-
        %a list of pairs of cores that don't share a cache

get_sharing_caches(List) :-
        %a list of pairs of cores that share a cache
```

6.4 Timing

This section describes the implementation of the timing algorithms. To measure cache and memory access latencies we used the ideas of *Yotov, et al.* [19]. However we will **not** focus on the explanation of the algorithms, the best description can be found directly in the paper [19].
As mentioned before the algorithms need information about the underlying system. For example it needs to know the associativity and capacity of every cache. The original version of the algorithm measures all these properties itself, but here we are in a better position, since we know that the SKB already stores (6.3.1) all this information. We also let the SKB determine the best way (5.2) to measure the time on the current processor.
Armed with this information we do the same as the authors of [19]: Start the clock, perform a number of accesses to memory, stop the clock and divide the execution time by the number of accesses to get the access time for one element. Unfortunately it is not as simple as that.
The problems are the caches themselves. When trying to measure the access time to the L2 cache for example, it is necessary that the data is not already present in the L1 cache. Thus we **need** an L1 cache miss. Similarly we need misses in all caches if we would like to measure the access time to main memory.

The idea, when trying to measure the access time to a specific cache level, is to perform memory accesses in a way that (1) cannot be anticipated by the pre-fetching mechanism of the processor and (2) produces cache misses in all higher levels of the cache. Refer to [19] for an explanation of how this can be achieved.
To measure the L1 latency we do not have to perform special access patterns to memory since it is the lowest cache level, apart from the registers. The challenge here is to keep the compiler from optimizing away the access operations. We use a macro which copies an instruction and thus we do not need to use a loop which would incur overhead. The macro we use is of the following form:

```
#define ONE(MACRO) MACRO
#define TWO(MACRO) ONE(MACRO) ONE(MACRO)
#define FOUR(MACRO) TWO(TWO(MACRO))
#define TEN(MACRO) FOUR(MACRO) FOUR(MACRO) TWO(MACRO)
#define HUNDRED(MACRO) TEN(TEN(MACRO))
#define THOUSAND(MACRO) TEN(HUNDRED(MACRO))
#define TENTHOUSAND(MACRO) TEN(THOUSAND(MACRO))
```

This allows us to measure the access time to the L1 cache in the following way:

```
uint64_t start, stop;
int i;

int source = 0;
volatile int *volat_ptr = &source;

register int reg;
reg = *volat_ptr;

start = get_timer_value(strategy);
THOUSAND(reg = *volat_ptr;);
stop = get_timer_value(strategy);

long double exec = (long double) get_timer_difference(strategy,start,stop)
                                        / (1000.0);
```

Note that we use helper - functions: "get_timer_value(strategy)" which returns a time value (not specified exactly, depends on the strategy (5.2)) the important point is that the "get_timer_difference(strategy)" method knows how much time has passed between two measurements (depending on the strategy).
To measure the access time to lower cache levels we need to perform the memory accesses in special ways [19]. The problem is that the access patterns depend on parameters of the systems that are unknown at compile time. This makes it impossible to completely avoid loops in our measurements. Hence we get inaccurate measurements due to loop - overhead.
To measure the access time to main memory we build the "void-pointer" data structure described in the paper by *Yotov et al.* [19]: an array of void pointers,

where every element contains the address of another element in the array. If we shuffle the addresses we can "follow" the pointers through the array and simulate a pseudo-random access pattern to memory:

```c
//every element stores it's address
for (int i = 0; i < size; i++) {
    *(void**)(elements + i) = elements + i;
}

//shuffle the addresses a little bit
for (int i = 0; i < size; i++) {
    const int other = rand() % size;

    void * temp = *(void**)(elements + i);
    *(void**)(elements + i) = *(void**)(elements + other);
    *(void**)(elements + other) = temp;
}

//traverse
volatile void *cursor = elements;
for (int i = 0; i < size; i++) {
    cursor = *(void **)cursor;
}

//avoid the compiler from optimizing away the loop:
if (NULL == cursor) {
    void *blocker = malloc(sizeof(void*));
    free(blocker);
}
```

Again we suffer from loop overhead, although in this case we think that the inaccuracy is negligible, since the access times to memory are much bigger compared to the amount of work that has to be done to control the loop (compare, add, jump).
Our current implementation also measures memory access times to other NUMA - Domains. To achieve this, we allocate memory on an another processor and then perform the same measurements as before.
After having measured the latencies we store the information in the following form:

latency(apicIdFrom, apicIdTo, cacheType, latency).

Where cacheType is a string identifying the cache level ("L1", "memory", etc.). And latency is the access time measured in cycles. ApicIdFrom denotes the core that executed the measurements and apicIdTo is the "target" core. If the datagatherer is running on core 0 and we have an access time of 377 cycles to the memory of core 9 we would store the information like this:

latency(0, 9, "memory", 377).

If we measure the access time to the L2 cache for example the apicIdFrom and the apicIdTo would be the same. However we plan to implement measurements

to remote caches as well in the future. Therefore the format is ready for this possible next step.
To access the data we provide the following queries:

```
get_latency_of_type(ApicIdFrom, ApicIdTo, Latency) :-
        latency(ApicIdFom, ApicIdTo, Type, Latency).

get_latency_l1_cache(ApicIdFrom, ApicIdTo, Latency) :-
        get_latency_of_type(ApicIdFrom, ApicIdTo, "L1", Latency).

get_latency_l2_cache(ApicIdFrom, ApicIdTo, Latency) :-
        get_latency_of_type(ApicIdFrom, ApicIdTo, "L2", Latency).

get_latency_memory(ApicIdFrom, ApicIdTo, Latency) :-
        get_latency_of_type(ApicIdFrom, ApicIdTo, "memory", Latency).
```

Overhead

To make our measurements more accurate we measure the overhead of our timing operations. Note that this has to be done **before** we measure the access times, so that we can subtract the overhead from all measured values.
The first thing we do is to measure the overhead of taking the time:

```
start = get_timer_value(strategy);
TENTHOUSAND(    a = get_timer_value(strategy);
                __asm(""::: );
                b = get_timer_value(strategy);
            )
stop = get_timer_value(strategy);

double overhead = (double) get_timer_difference(strategy, start, stop)
            / (double) (10000+1);
```

Of course the problem is "circular" since we suffer from overhead when we try to measure the overhead of the measurement. Nevertheless we think that the n is big enough for a decent value.
We store the data in the following format

timing_overhead(apicId, overhead).

To access the information, we provide the following query:

```
get_timing_overhead(Apic, Overhead) :-
        timing_overhead(Apic, Overhead).
```

To measure the loop overhead we execute an empty loop. The important thing however, is that it is not optimised away by the compiler. That is why we use the following blocker (we use gcc 4.5.2):

```
for (int i = 0; i < n; i++) {
        __asm("":::);
}
```

which will then be translated into something that contains only one jump, compare and increment instruction. We store the value in the following format:

$$\text{loop\_overhead}(\text{apicId}, \text{overhead}).$$

The information can be retrieved with the following query:

```
get_loop_overhead(Apic, Overhead) :-
        loop_overhead(Apic, Overhead).
```

7 Evaluation

The datagatherer performs all the hardware specific tasks to gather information about the system the OS is currently running on. It fills the SKB with useful information that other applications can use for their benefit. This document covered the design (5) and implementation (6) of the Intel datagatherer and introduced a format to store the information.
The complete program took about 1600 lines of C code to implement. In addition we produced about 180 lines of Prolog code.

7.1 Test Setup

We tested the datagatherer on two different machines. The first one is an Intel Xeon X5355 (Clovertown) and an Intel Xeon L7555 (Beckton) with the following specifications:

	Clovertown	Beckton
Cores	4	32
Threads	4	64
Clock Speed	2.66GHz	1.86GHz
Biggest Cache	L2: 8MB	L3: 14MB

After Barrelfish has booted it starts the SKB, and the datagatherer which instantiates itself on every core in the system.

7.2 Runtime

In this section we give an overview of the runtime of the datagatherer. It collects (6.2) general information about the hardware, the caches (6.3) and measures (6.4) access times to the caches and to memory. As one might expect it takes quiet a long time to measure all the access times. The first part however, in which the hardware is queried with the CPUID instruction, executes very fast. We executed the following tasks:

Task 1: Query the hardware

Task 2: Task 1 and measure cache latencies

Task 3: Task 2 and measure the access time to local memory

Task 4: Task 3 and measure the access time to remote memory

and found the following execution times on our Beckton:

	Execution Time
Task 1	$\sim$ 0.08s
Task 2	$\sim$ 6s
Task 3	$\sim$ 83s
Task 4	$\sim$ 45 minutes

The execution time of Task 4 is the maximum we could find in our system. That means we looked at a processor that executed 31 remote measurements.
These numbers are by no means very accurate. The reason why they are presented is to give the reader a feeling of how long it takes to perform certain tasks. Note that the Task 1 (querying the hardware) can be done in parallel on all datagatherers, while all the measurements have to be done sequentially to avoid bus contention.
It is immediately clear that we cannot perform all latency measurements when the OS starts up. It would take far too long. The hardware however can be queried without problem on every startup. In our opinion it is acceptable to let every datagatherer perform the cache measurements as well. On the Intel Beckton we found that is takes about three minutes for all datagatherers to finish, which is still reasonable.
As indicated above the main problem is the time it takes to measure the memory access latencies. It gets even worse when we measure the access times to all remote memory banks. This is simply not feasible. An approach could be to postpone these measurements to the future and let the SKB pick a good moment to do them. We do not want to shift the problem to another place, but we believe the SKB is in a better position to do this decision. It might for example be possible to perform one specific measurement lazily (when the fact is requested). The program would then have to be stalled for about 80 seconds. In special circumstances this might be tolerable, however in most scenarios this is of course also impossible and in fact useless, since there is obviously no gain in performance.
The measurements could be executed randomly, meaning that the SKB chooses a time slot to perform a measurement and eventually, if the OS runs long enough, all access times are stored.
There is one simplification that can be applied in the datagatherers that greatly reduces the amount of work that has to be done. If we have a pair of processors, say A and B, we do not measure the latency from $A \to B$ and from $B \to A$, but we do only one direction and assume that the other one would yield about te same result. This reduces the amount of work by a factor of two. However, if we have n logical processors we still need to do about

$$\frac{n(n-1)}{2} = \frac{n^2 - n}{2} \in \mathcal{O}(n^2)$$

measurements, which is quiet a lot, assuming that one measurement takes about 80 seconds.
The big problem with the simplification described above is that the assumption, that $A \to B = B \to A$, is in general not true. Therefore when we do the work properly we end up with twice as much measurements:

$$n(n-1) = n^2 - n \in \mathcal{O}(n^2)$$

7.3 Storing Data

In this section we take a closer look at our data format. In section 6.2.2 we describe how available features are stored. There is an inherent problem with storing the features of a processor: they are processor / vendor specific. Some features might be used for multiple processors / manufacturers while a great number of them cannot. For example the "vmx" (virtual machine extensions) feature seems to be the name of an Intel specific technology. It is possible to map this instruction to an abstract concept (namely the support for virtualisation). This allows the whole complexity to be pushed into the SKB, because this information is enough for applications. The SKB alone has to know how the "support for virtualisation" is actually implemented on a specific processor.
Let us now look at the different facts that are collected by the datagatherer:

- (1) vendor($\underline{\text{apicId}}$, vendorName).
- (1) prefetching($\underline{\text{apicId}}$, value).
- (2) thermal_power($\underline{\text{apicId, featureName}}$, value).
- (2) monitor_mwait($\underline{\text{apicId, featureName}}$, value).
- (2) performance_monitor($\underline{\text{apicId, featureName}}$, value).
- (2) dca_parameter($\underline{\text{apicId, featureName}}$, value).
- (2) address_size($\underline{\text{apicId, memoryType}}$, value).
- (3) cpu_thread($\underline{\text{apicId}}$, packageId, coreId, $\underline{\text{threadId}}$).
- (4) feature($\underline{\text{apicId, featureName}}$).
- (4) cache_detail($\underline{\text{apicId, feature}}$).
- (5) cache($\underline{\text{apicId, level, type}}$, size, assocN, assocType, lineSize).
- (5) cache_identifier($\underline{\text{apicId, cacheType}}$, threadsPerCache, cacheId, $\underline{\text{level}}$).
- (6) latency($\underline{\text{apicIdFrom, apicIdTo, type}}$, latency).

The underlined entries represent the $\underline{\textbf{keys}}$ to the relations.
Obviously we are in 1NF (3.6), because we only store atomic values. The first two relations of category (1) are in Boyce-Codd NF (3.6) since "apicId" is the superkey of both relations. For the relations in category (2), (3) and (6) we also have functional dependencies of the form $\alpha \rightarrow \beta$ where α is a superkey and thus we are in Boyce-Codd NF.
In category (4) we have trivial functional dependencies and therefore are also in Boyce-Codd NF. Category (5) is also in Boyce-Codd NF. It is important to see that both keys are minimal, there is no subset which describes one specific entry in the SKB. In particular, the "cacheId" is not unique. "Level" and "apicId" cannot be used as a key either, since there can be multiple caches at the same level (instruction and data caches).

7.4 Timing Overhead

On the Intel Clovertown we found that the overhead of the time measurement (5.2.1) is about 500 cpu cycles. For measuring the access latency to the L1 cache we execute 10000 instructions. According to:

$$E = \frac{500\text{cycles}}{10000}$$

we get an error of about 0.05 cycles. To measure the L2 cache, we made 16386 accesses to memory which results in an error of about 0.03 cycles.
On another machine, an Intel Xeon L7555 (Beckton), we found the overhead of the time measurement to be about 105 cycles which results in an error of about 0.0105 cycles for the L1 measurement, about 0.006 cycles for the L2 measurement and about 0.0001 cycles for the L3 measurement, respectively.

8 Conclusion

We designed (5) and implemented (6) a datagatherer for the Barrelfish OS. We demonstrated a way to efficiently gather information about the hardware the OS is currently running on. The datagatherer is a useful contribution to the Barrelfish OS, since it provides a solid basis that makes it possible for a wide variety of programs to apply optimisations that are very hardware specific.
The information that is made available could for example be used by algorithms to adapt themselves to cache sizes, etc, while multi-threaded programs can choose cores that support their needs (two cores that share a cache for example).
The information is stored in a format that is independent from hardware vendors, thus making it possible for the (Intel -) datagatherer, introduced in this document, to work together with datagatherers for other hardware without changing its implementation. The combination of different datagatherers, each of which is specialised to a specific kind of hardware, makes the SKB ready for future systems with heterogeneous cores. The data is stored in a generic format that allows application to run mostly oblivious of the specific hardware manufacturer, although the information could be accessed if it was needed.

8.1 Future Work

There are several things we would like to implement in the future. First of all we can measure more access times, for example to caches of other processors. This information allows for even better adaptation of algorithms to the hardware.
As mentioned in 7.2 we can still improve the way in which memory access times are collected. It takes a very long time to execute all measurements and it really is necessary to do this work gradually. For Intel hardware, it would be very interesting and rewarding to analyze the QickPath Interconnect (QPI). This information is very useful since it basically provides a “map” of the network, i.e. the network topology. Programs can use this knowledge to further optimise the core-selection since it is easy to choose cores that are “close together”. This can already be done approximatively by looking at the inter - core latencies. However the QPI network is extremely fast and it would be useful to make this information available.
Another task is to change existing programs (of the Barrelfish OS) in such a way that they actually utilise the information that is exposed by the datagatherers.

9 Glossary

Barrelfish
A research operating system released by ETH Zurich that focuses on multi- and many-core systems. Find more on the website: `http://www.barrelfish.org`

CPU Driver
A mini-kernel that runs on every processor. Also take a closer look at "Multikernel"

CPUID Instruction
An instruction that is used to query the hardware. Check section 3.5 for an introduction, or look at the online documentation [10].

Datagatherer
A program that is executed on every core to gather information about the hardware (e.g. cache size). Check section 3.4 for an introduction.

System Knowledge Base (SKB)
Stores all kinds of information that can be useful for application (e.g. L1 cache hit latency). Applications can query this database using Prolog queries. Refer to section 3.2 for more details.

Non Uniform Memory Access (NUMA)
Memory can be physically partitioned into smaller bits. Several cores are grouped around one such "element of memory" that can be accesses fairly fast. But when accessing remote memory, the access time becomes bigger (thus the name, non uniform memory access).

NUMA - Domain
The cores that are grouped around a part of the memory form a NUMA - Domain.

Multikernel
Architectural model by *Bauman, et al.* [4]: One mini-kernel, called cpudriver,

runs on every processor. There is no shared state between the kernels, a consistent state is achieved with message passing. All these kernels work together like a distributed system and form an OS.

Prolog
A logic programming language that is used to store and retrieve information from the SKB. Section 3.3 gives a short introduction to the language.

Intel Xeon X5355 (Clovertown)
4 cores, 4 threads, 2.66 GHz, 8MB L2 cache, `http://ark.intel.com/products/28035/Intel-Xeon-Processor-X5355-8M-Cache-2_66-GHz-1333-MHz-FSB`

Intel Xeon L7555 (Beckton)
8 cores, 16 threads, 1.866 GHz, 24MB L3 cache, `http://ark.intel.com/products/46494/Intel-Xeon-Processor-L7555-24M-Cache-1_86-GHz-5_86-GTs-Intel-QPI`

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[2] Krzysztof R. Apt and Marg G. Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.

[3] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[5] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. why isn't your os? In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 12–12, Berkeley, CA, USA, 2009. USENIX Association.

[6] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. Learn prolog now! `http://www.learnprolognow.org/`.

[7] Cisco. ECL^{i}PS^e. `http://www.eclipse-clp.org`.

[8] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. FFT program generation for shared memory: SMP and multicore. In *Supercomputing (SC)*, 2006.

[9] Intel. Intel 64 architecture processor topology enumeration. `http://software.intel.com/en-us/articles/intel-64-architecture-processor-topology-enumeration`.

[10] Intel. Intel processor identification and the cpuid instruction. `http://www.intel.com/content/www/us/en/processors/processor-identification-cpuid-instruction-note.html`, May 2012.

[11] Alfons Kemper and André Eickler. *Datenbanksysteme, Eine Einführung*. Oldenbourg Wissenschaftsverlag GmbH, 7. edition, 2009.

[12] William Kent. A simple guide to five normal forms in relational database theory. *Commun. ACM*, 1983.

[13] Simon Peter, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe. Early experience with the barrelfish os and the single-chip cloud computer.

[14] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[15] Jinuk Luke Shin, Dawei Huang, Bruce Petrick, Changku Hwang, Kenway Tam, Alan Smith, Ha Pham, Hongping Li, Timothy Johnson, Francis Schumacher, Ana Sonia Leon, and Allan Strong. A 40 nm 16-core 128-thread sparc soc processor. *J. Solid-State Circuits*, 46(1):131–144, 2011.

[16] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[17] R. Clint Whaley, Antoine Petitet, and Jackj Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 2001.

[18] Jianxin Xiong, Jeremy Johnson, Robert W. Johnson, and David Padua. SPL: A language and compiler for DSP algorithms. In *Programming Languages Design and Implementation (PLDI)*, pages 298–308, 2001.

[19] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic measurement of memory hierarchy parameters. *SIGMETRICS Perform. Eval. Rev.*, 33(1):181–192, June 2005.