How to Optimize the Scalability & Performance of a Multi-Core Operating System

Architecting a Scalable Real-Time Application on an SMP Platform

IntervalZero

Overview

When upgrading your hardware platform to a newer and more powerful CPU with more, faster cores, you expect the application to run faster. More cores should reduce the average CPU load and therefore reduce delays. In many cases, however, the application does not run faster and the CPU load is almost the same as for the older CPU. With high-end CPUs, you may even see interferences that break determinism. Why does this happen, and what can you do about it?

The answer: build for scalability. Unless an application is architected to take advantage of a multicore environment, most RTOS applications on 1-core and 4-core IPCs will perform nearly identically (contrary to the expectation that an RTOS application should scale linearly and execute 4 times faster on a 4 core IPC than it does on a 1 core IPC.) Without a scalable system, 3 of the 4 cores on the 4 core system will not be utilized. Even if the application seeks to use multiple cores, other architectural optimizations involving memory access, IO, caching strategies, data synchronization and more must be considered for the system to truly achieve optimal scalability.

While no system delivers linear scalability, you can work to achieve each application's theoretical limit. This paper identifies the key architectural strategies that ensure the best scalability of an RTOS-based application. We will explore CPU architectures, explain why performance does not get the expected boost with newer or more powerful cores, describe how to reduce the effects of the interferences, and provide recommendations for hardware modifications to limit bottlenecks .

Introduction

This paper, written for RTOS users, addresses a system where both real-time and non-real-time applications run at the same time. To keep determinism in the real-time applications, they ideally should not share any hardware with the non-real-time applications. But at the same time, it is helpful to have memory spaces and synchronization events available to both sides.

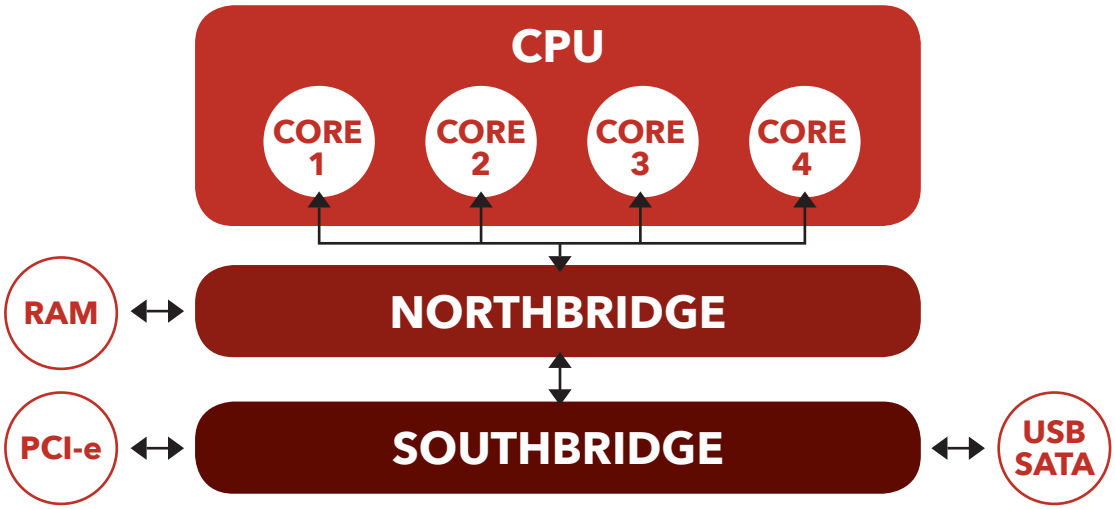
However, it is not possible to achieve both. Either you have a dedicated real-time computer but must rely on a bus protocol to exchange data with the non-real-time applications, or you have both on the same machine but they will share the CPU bus and cache. Nowadays CPU cores are much faster than memory and I/O access, so the interferences come from competition in the access of these resources.

There is another important thing to consider when using multiple cores. The different threads in an application usually share variables, so access to these variables has to be synchronized to ensure the values are consistent. The CPU will do this automatically if it is not handled in the code, but as the CPU does not know the whole program, it will not handle it optimally and this will create many delays. These delays are the reason that an application will not necessarily run faster on two cores than on one.

This paper will first examine the CPU architecture relating to caches, memory and I/O access. Then we will explain how threads interact and how program design can help improve performances with multiple cores. Finally, we will give examples of practical problems and what can be done to solve or minimize them.

Most of the technical information in this paper is based on the excellent paper What Every Programmer Should Know About Memory by Ulrich Drepper at Red Hat. We recommend reading that paper if you have the time.

1. CPU Architecture



1.1. Traditional Architecture: UMA (Uniform Memory Access)

In this model, all the cores are connected to the same bus, called Front Side Bus (FSB), which links them to the Northbridge of the chipset. This RAM and its memory controller are connected to this same Northbridge. All other hardware is connected to the Southbridge which connects to the CPU through the Northbridge.

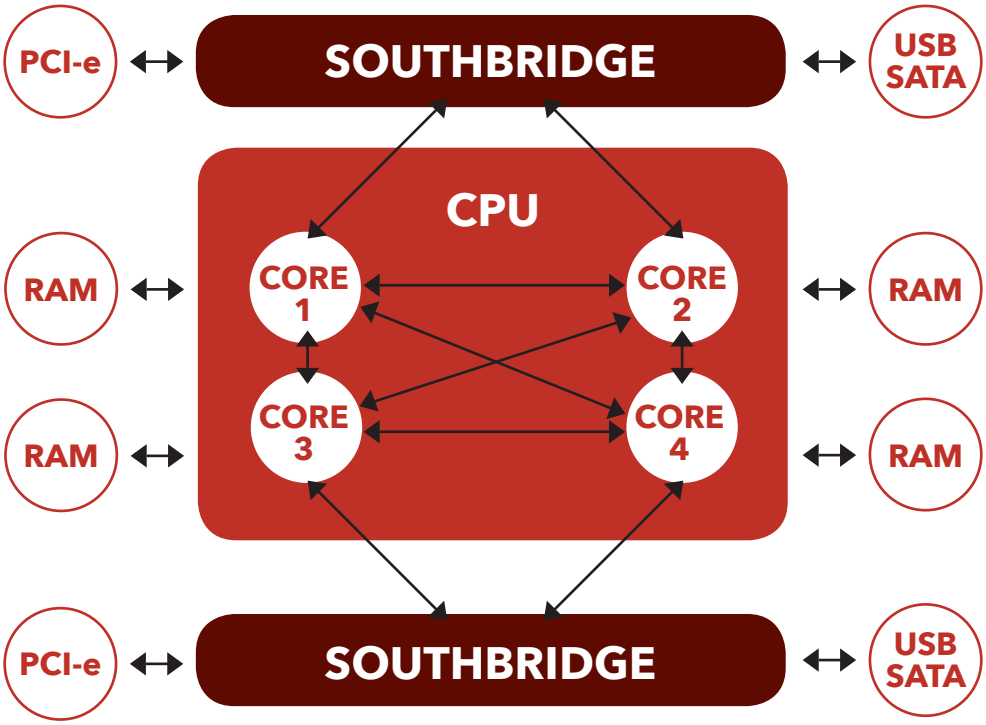
From this design we can see that the Northbridge, Southbridge and RAM are resources shared by all the cores and therefore by both real-time and non-real-time applications. Additionally, RAM controllers only have one port, which means only one core can access the RAM at a time.

The CPU frequencies have increased steadily for many years without becoming more expensive but it has not been the same for memory. Persistent memory access (such as hard drives) is very slow, so RAM was introduced to allow the CPU to execute code and access data without having to wait for the hard drive accesses.

Very fast Static RAM is available, but as it is extremely expensive it can only be used in low amounts in standard hardware (a few MB). What we commonly call RAM in computers is Dynamic RAM, which is much cheaper but also much slower than Static RAM. Access to Dynamic RAM takes hundreds of CPU cycles. With multiple cores all accessing this Dynamic RAM, it is easy to see that the FSB and RAM access are the biggest bottlenecks in the traditional architecture.

1.2. NUMA (Non-Uniform Memory Access) Architecture

To remove the FSB and RAM as bottlenecks, a new architecture was designed with multiple Dynamic RAM modules and multiple busses to access them. Each core could possibly have its own RAM module. The Southbridge to access I/O can also be duplicated so different cores could use different busses to access the hardware. For real-time applications, this would have the added advantage of no longer sharing these resources with non-real-time applications.



Originally NUMA was developed for the interconnection of multiple processors, but as processors now have more and more cores, it has been extended and used inside processors.

The NUMA design introduces new problems, as variables are only located in a single RAM module while multiple cores may need to access them. Accessing variables attached to a foreign core may be much slower and applications should be developed specifically for this architecture to use it properly. We would recommend only using this architecture for applications developed for it, but when the number of cores on a system exceeds four, the FSB of the UMA architecture gets very easily overloaded and may cause even more delays. Therefore, NUMA will be the architecture for larger machines.

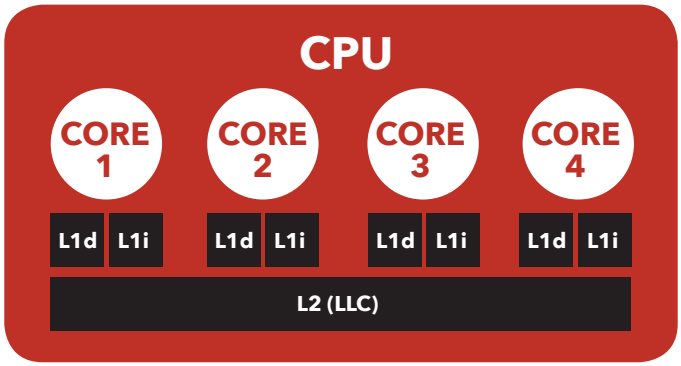
To gain the advantages of NUMA without its issues, some machines are made with nodes of processors sharing a RAM module. In that case applications only using cores inside a single node would not see the effects of NUMA and work normally.

We will not go into more details on this architecture as it is not supported by RTX at the moment.

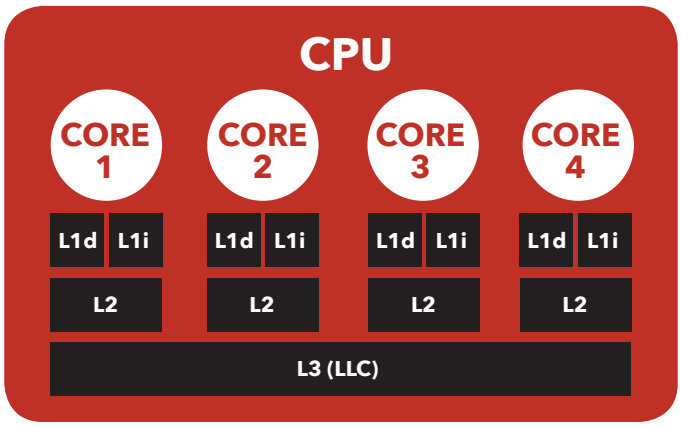
1.3. Memory and Caches

As mentioned previously, Dynamic RAM access is slow for the CPU (hundreds of cycles on average to access a word) compared to access to Static RAM. Therefore, CPUs now include Static RAM used as cache and organized in multiple levels. When a core needs to access data, that data will be copied from the main memory (Dynamic RAM) into its closed cache so it can access it multiple times faster.

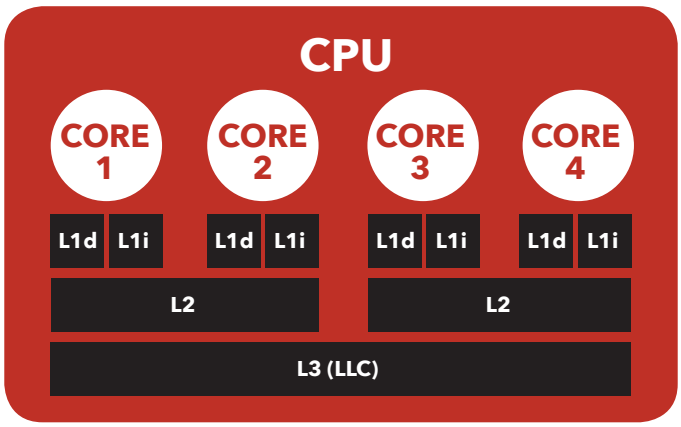
QUAD CORE CPU WITH TWO LEVELS OF CACHE



QUAD CORE CPU WITH THREE LEVELS OF CACHE, LEVEL 2 IS EXCLUSIVE



QUAD CORE CPU WITH THREE LEVELS OF CACHE, LEVEL 2 IS SHARED BY CORE NODES



The first level of cache (L1) has separate areas for instructions (the program code) and data (the variables); other levels are unified. Higher levels of cache are larger and slower than the first level and may be exclusive to a core or shared by multiple cores. The largest cache, also called Last Level Cache (LLC), is normally shared by all cores. Average access times for each cache level given in CPU cycles is given in the table below.

CACHE LEVEL	AVERAGE ACCESS TIME
LEVEL 1	~ 3
LEVEL 2	~ 15
LEVEL 3	~ 20
MAIN MEMORY	~ 300

As you can see, performance takes a huge hit any time the CPU has to wait for the main memory to be accessed because the data is not available in the cache. This is called a "cache miss". The main memory data access is much faster if it is done in bulk or in order. But the data and instruction access in a program is rarely random, so the CPUs will try to predict which memories will be used next and load them to the cache in advance. This technique is called prefetching and improves performance significantly (~90% delay reduction).

To predict which data should be in the cache, processors rely on two principles: temporal locality and spatial locality.

- Temporal locality means that variables and instructions are usually accessed multiple times in a row. This is true especially with loops and variables local to a function.
- Spatial locality means that variables defined together are usually used together and the next line of code most likely contains the next instruction to execute.

Temporal locality is the reason for having caches. Copying the data to a local buffer before using it is only relevant if it will be accessed multiple times. To take advantage of spatial locality and the fact that RAM is accessed faster in bulk, data is not requested and transferred in bytes but in cache lines which are normally 64 bytes long. Also, the CPU will usually prefetch the next line automatically. The work of the programmer, detailed in section 2, is to make the data and instruction order as predictable as possible so that the prefetch works efficiently.

1.4. Bottlenecks

Because caches are expensive they are usually small, so not all the data and instructions related to an application can be in a cache. Also, it is shared by all the applications running on the cores attached to this cache. This means that when an application or thread is loading too much data, it will evict older data that another thread or application may still want to use and need to reload. The more code that is executing on the core, the more likely instructions will be evicted when thread switching. This fight for cache is called "memory contention".

The LLC is shared by both real-time and non-real-time applications when the system is a single socket or a multiple socket system is configured with a socket having both real-time and non-real-time cores. As a result, non-real-time applications can affect the performance of the real-time application when they use a large amount of memory, by running an HD video for example. If the amount of data used by an application or thread is small, it may be possible to choose a CPU with enough cache to keep that whole data in the exclusive caches where it will not be affected by other applications.

The FSB which accesses the main memory is very slow compared to the CPU, so heavy data loading from a single core could use the whole bandwidth alone. This is called "bus contention" and will start being visible when the CPU has four or more cores. As this bus is the real bottleneck, it is usually better to spend money on a faster RAM and Chipset bus than on a faster CPU. A faster CPU will usually just wait longer.

1.5. Data Synchronization

The main reason why an application does not run faster on two cores than one is data synchronization. Code serialization can also play a role in execution latency. As mentioned previously, cores always access data through their lowest level of cache which is exclusive. This means that if a variable is accessed by two cores, it must be present in the cache of both cores, but when its value is modified it has to be updated in the cache of both cores. The CPU must ensure data consistency for the whole system which can cause huge delays, generally as a result of one core needing to snoop data in the cache of another core to ensure data integrity.

To maintain this consistency the CPU uses the MESI protocol, which defines the state of a cache line.

- **MODIFIED**: The value has been modified by this core so this is the only valid copy in the system.
- **EXCLUSIVE**: This core is the only one using this variable. It does not need to signal changes.
- **SHARED:** This variable is available in multiple caches. Other cores should be informed if it changes.
- **INVALID:** No variable has been loaded or its value was changed by another core.

The status of each cache line is maintained by each core. To do this, they have to observe all data requests to the main memory and inform other cores that they already have a variable being read or have modified its value. Every time a core wants to access a variable that is modified in the cache of another core, the new value has to be sent to the main memory and the reading core. Access to this value becomes as slow as if there were no cache. If there is a variable written by one core often and read by another, here is what will happen:

ACTION	CORE 1 STATUS	CORE 2 STATUS
Core 1 reads the value	EXCLUSIVE	INVALID
Core 2 reads the value	SHARED	SHARED
Core 1 modifies the value	MODIFIED	INVALID
Core 2 reads the value	SHARED	SHARED
Core 1 modifies the value	MODIFIED	INVALID
Core 2 reads the value	SHARED	SHARED

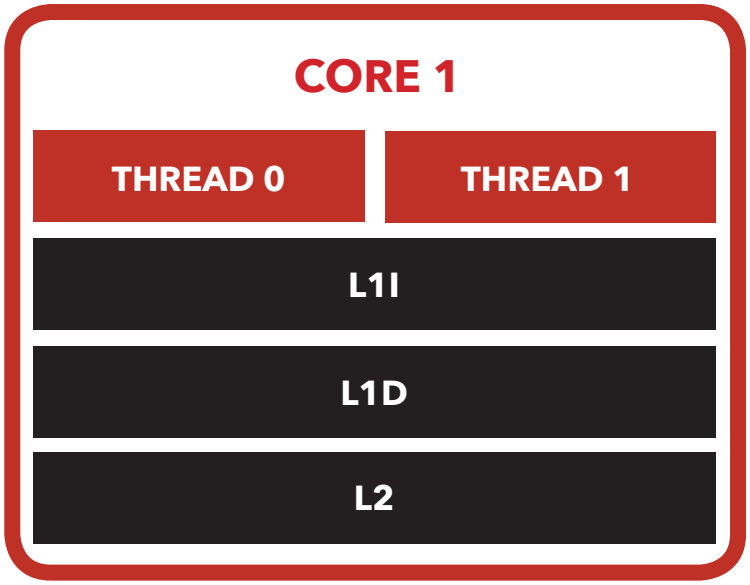
In this case the Core 2 has to read the value from the main memory every time, which removes the advantage of the cache. And the Core 1 has to send a "Request For Ownership" (RFO) on the FSB every time it modifies the value and then update the main memory every time the Core 2 requests the value.

Access to this value will be much slower when the two threads are on different cores compared to both threads running on a single core and it will add traffic to the FSB.

There is much less of a problem for instructions because they are normally read-only. In this case there is no need to know how many cores are using it. Self-modifying code exists but it is very dangerous and rarely used, so we will not address it here.

1.6. Multi-Core and Hyperthreading

Multiple cores and multiple threads on a single core seem to have the same use but the way they share resources is very different. As a result, the way they should be used is almost opposite.



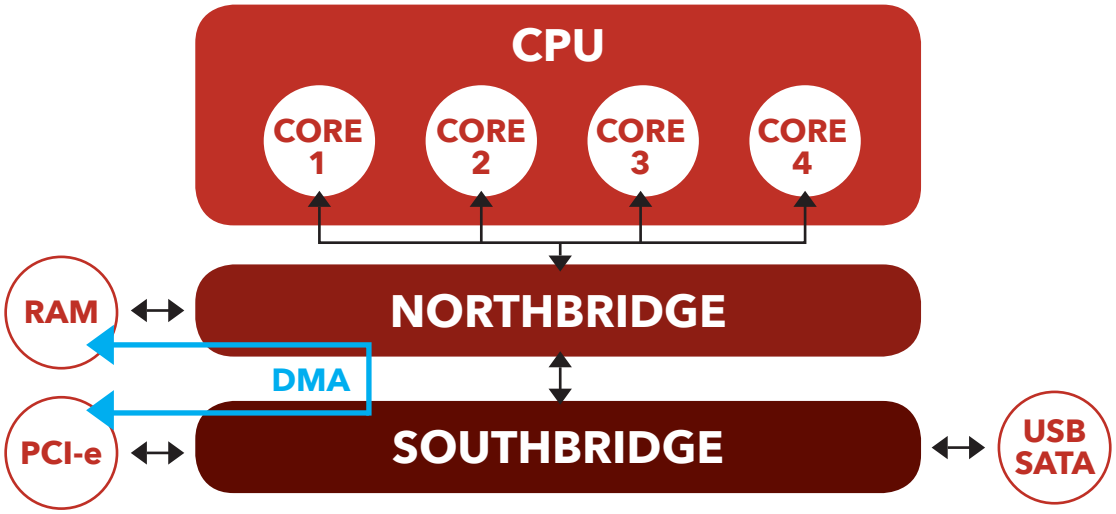
With multiple cores, the level 1 cache is duplicated and each core has its own. This means that more cache is available on the system but they need to be synchronized.

With hyperthreading, the two threads share the same level 1 cache, so in the worst case would only have half of it available.

So, with multiple cores programmers must limit the amount of shared data between the threads on each core to avoid synchronization delays. With hyperthreading, the level 1 cache is shared between the hyper threads. This means if the data used by each thread is different it will cause cache contention and data will have to be loaded from the main memory much more often. In this case performance will be improved only if independent operations are made on the same data set. This is usually a special situation, so in most cases hyperthreading will not improve performance and we suggest disabling it. Also, since both the core and level 1 cache are shared between the two hyper threads, both of them should be used by real-time applications or non-real-time applications.

1.7. DMA (Direct Memory Access)

Access to the I/O, which can be PCI-e, USB or any other type, is controlled by the CPU instructions. This means that when a device such as a NIC signals updated data with an interrupt, the CPU has to query for the data and send it to the main memory. This adds a lot of unnecessary load on the FSB. For high-speed busses, the Direct Memory Access (DMA) feature was developed. Using DMA a device will signal the CPU that data was updated and directly send this data to the main memory without needing any action from the CPU.



If the CPU plans to immediately use this data, there is a new feature that can be used called Direct Cache Access (DCA) where the data would be copied to both the main memory and CPU cache.

2. Memory & Multicore Programming

2.1. Caching Methods

Caching is completely controlled by the processor and cannot be modified by the programmer. But the way it is implemented may have a huge impact on the program performance. Data available in the different caches may be different or the data in lower-level caches may be duplicated in higher-level caches. Having this data duplicated makes the higher-level cache seem smaller, but if two cores access the same variable shared by that higher-level cache, then they can use it to synchronize values instead of going all the way to the main memory.

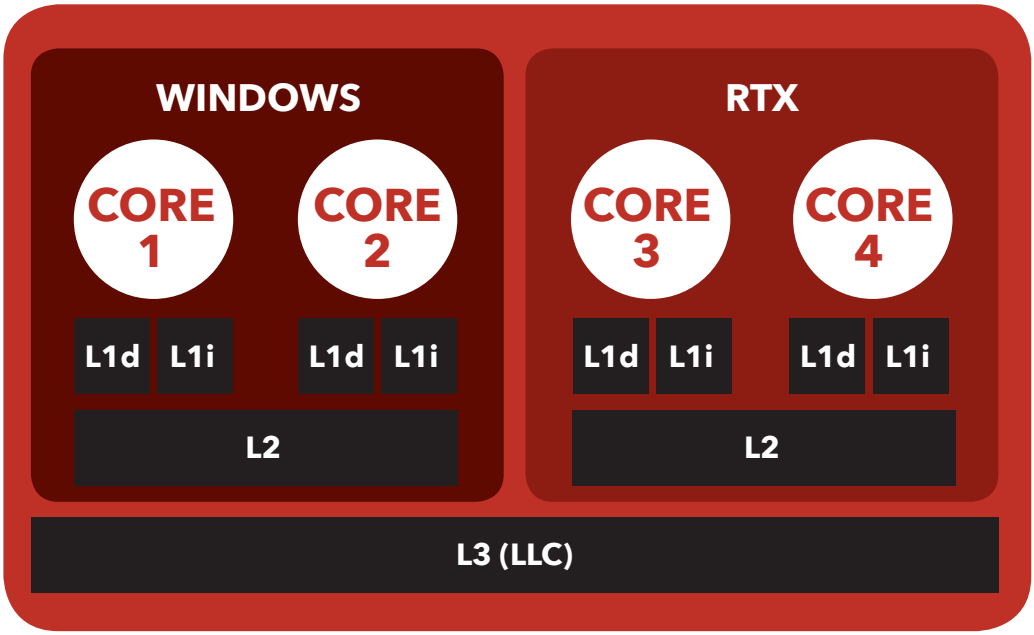
We can assume that the cache is always full as this will be true once the computer has been running for a few minutes. So, any data added to the cache will cause another cache line to be "evicted", which means copied back to the next level of cache which will in turn send a value back to the main memory if it did not already contain the evicted value. By default, the oldest used value in the cache will be evicted, although some newer processors have more complex calculations to choose which value to evict. Processors have memory management instructions that can be used to force or bypass these cache features, but using these functions is very hardware- and application-specific and requires development time. These functions will not be described here but are explained in Ulrich Drepper's paper referenced in the introduction.

2.2. Exclusive and Shared Caches

As described in section 1.3, there are multiple cache architectures available which impact performance. The selection of which core each thread should run on should depend on how the threads interact and what caches are available.

Ideally, threads that share many variables should run on the same core and threads running on different cores should not share variables. But if this was possible, there would only be small applications running on a single core. Bigger applications require multiple cores to run different modules and still need to share data and synchronize modules.

Programmers therefore need to identify which variables are shared or not and try to keep as many local variables in the exclusive cache while having shared caches to contain shared variables. In the specific case of RTX where there are two operating systems each with their own cores, an ideal situation would be to have three levels of cache: the level 1 exclusive to each core, the level 2 separated in RTX core cache and Windows core cache, and the last level cache shared by all cores. This way the Windows applications cannot pollute the RTX level 2 cache and threads on the different RTX cores can share variables without relying on hardware which is shared with the non-real-time space. This ideal situation, however, is only valid when the most commonly used variables in the RTX applications do not exceed the size of the level 2 cache.



2.3. Optimizing Variable Declaration

To take advantage of the multi-core and cache optimizations, different variable types have to be identified and handled differently. The different variable types are:

- Variables used by a single core. These variables should only appear in one level 1 cache. While they can be evicted to L2/LLC, they cannot appear in multiple L1 caches.
- Real-only variables (initialized at the beginning and then never changed). These variables can be shared by multiple cores without performance issues.
- Mostly read-only variables.
- Often modified variables. Variables that are often modified by multiple cores or by a core while in a shared state will have slow access, so they should be grouped together to avoid interfering with other variables.

There are compiler definitions that can ensure alignments and explicitly indicate the type of variables but improvements can be achieved just by declaring the variables properly.

- Variables are accessed by cache lines of 64 bytes, so it is best to make sure only one type of variable is available in a cache line. To achieve this, variables can be grouped in structures whose length is a multiple of 128 bytes.
- The total size of the structures should be as small as possible and variables will be aligned in structures (we can assume a 64-bit alignment on 64-bit systems) so smaller variable types should be grouped together. For example, two int or four short.
- Prefetching will load the next cache line so the first variable to be used should be at the beginning of the structure and variables should be declared in the order they are used.
- "Mostly read-only" and "Often modified" variables that are consumed together should be grouped together so that they can all be updated in a single operation.

If these rules are not followed you may have an unexpected situation called "false sharing". This happens when a variable that should be read-only or consumed by a single core is marked invalid because it is on the same cache line as a variable modified by a different core. In that case access to the first variable will be slow even though it should not be. Having the different types of variables in different cache lines ensures this situation does not happen.

Access to shared and often modified variables may be slow and even subject to interference if the FSB is overloaded. These variables should not be accessed by threads which are very deterministic and require small jitters. To achieve this, it may be useful to create core specific relay variables that are accessed by the highly deterministic thread and let a lower priority thread synchronize the values with the shared variables.

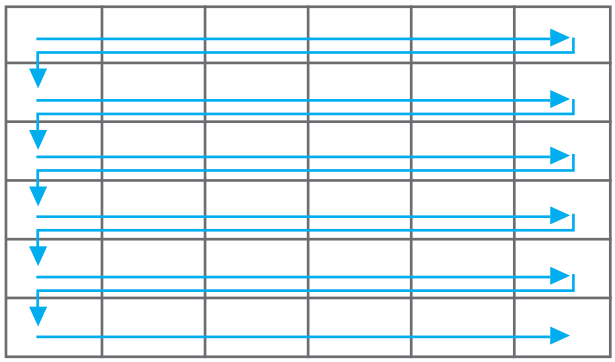
2.4. Optimizing Variable Access

When work is done on big data sets which exceed the cache size, it may be useful to write the code in a way that optimizes cache use. These operations could be picture analysis or other operations on big matrices. This section should only be considered if the matrix size is too big to fit in the cache. Since the data will have to load from the main memory, it should be consumed in a way that takes advantage of cache and RAM technologies:

- If possible, the data set should be broken into smaller sets that fit in the cache and all operations on a single set should be done before moving to the next set.
- The data should be accessed in the order it is defined in the memory so that prefetching reduces the loading time.

We will use the multiplication of two square matrices A and B, each with 2000 rows and columns, as an example to show possible modifications to the code.

A matrix is defined in the memory as an array of arrays. So, variables are organized in the following way:

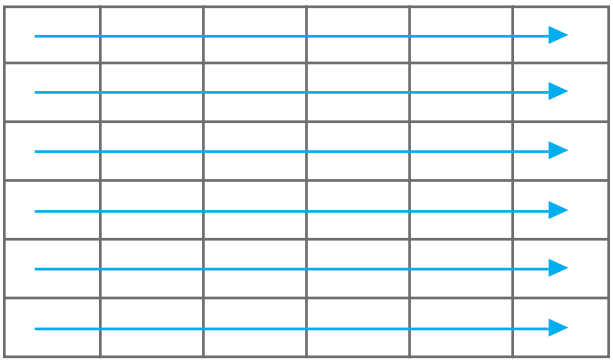


The standard code to accomplish the multiplication would be very simple and use 3 for loops:

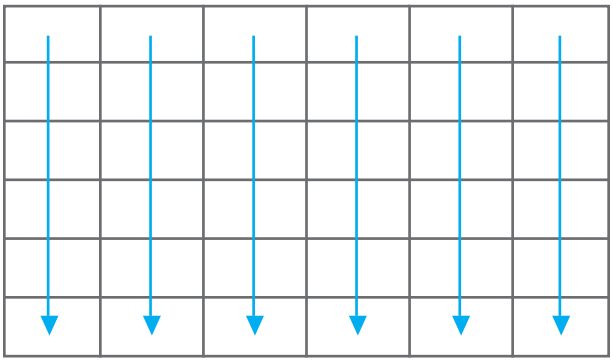
```
for (int i = 0; i < 2000; i++) {
        for (int j = 0; j < 2000; j++) {
                for (int k = 0; k < 2000; k++)
                        Result[i][j] += A[i][k] * B[k][j];
        }
}
```

With this code the matrices are consumed in different ways:

A:



B:



With this simple logic, the data in the first matrix is processed only once and in the order it is located in the memory. But the data in the second matrix is loaded many times and in an order which looks random to the processor.

In such cases the programmer should try to break the calculations in smaller datasets that fit in the L1d cache and try to finish using a dataset before moving to the next one.

```
int SetSize = DataSetSize / CellDataSize; // 64 / 8
int Iteration = 2000 / SetSize;
for (int i = 0; i < Iteration; i+= SetSize) {
        for (int j = 0; j < Iteration; j+= SetSize) {
                for (int k = 0; k < Iteration; k+= SetSize) {
                        for (int i2 = 0; i2 < SetSize; i2++) {
                                for (int j2 = 0; j2 < SetSize; j2++) {
                                        for (int k2 = 0; k2 < SetSize; k2++)
                                                Result[i][j+SetSize*i2+j2] +=
                                                        A[i][k+SetSize*i2+k2] * B[k][j+SetSize*k2+j2];
                                }
                        }
                }
        }
}
```

To simplify the equation, pointers can be used to represent the matrices:

```
int SetSize = DataSetSize / CellDataSize; // 64 / 8
int Iteration = 2000 / SetSize;

for (int i = 0; i < Iteration; i+= SetSize) {
        for (int j = 0; j < Iteration; j+= SetSize) {
                for (int k = 0; k < Iteration; k+= SetSize) {
                        for (int i2 = 0; i2 < SetSize; i2++) {
                                R2 = &Result[i][j + SetSize * i2];
                                A2 = &A[i][k + SetSize * i2];
                                for (int j2 = 0; j2 < SetSize; j2++) {
                                        for (int k2 = 0; k2 < SetSize; k2++) {
                                                B2 = &B[k][j + SetSize * k2];
                                                R2[j2] += A2[k2] + B2[j2];
                                        }
                                }
                        }
                }
        }
}
```

To simplify the equation, pointers can be used to represent the matrices:

```
for (int i = 0; i < Iteration; i+= SetSize) {
        for (int j = 0; j < Iteration; j+= SetSize) {
                for (int k = 0; k < Iteration; k+= SetSize) {
                        for (int i2 = 0; i2 < SetSize; i2++) {
                                R2 = &Result[i][j + SetSize * i2];
                                A2 = &A[i][k + SetSize * i2];
                                for (int k2 = 0; k2 < SetSize; k2++) {
                                        B2 = &B[k][j + SetSize * k2];
                                        for (int j2 = 0; j2 < SetSize; j2++)
                                                R2[j2] += A2[k2] + B2[j2];
                                }
                        }
                }
        }
}
```

Such code modifications can reduce the calculation time by 75%, which can make a big difference. But as they make the code more complex and introduce new variables, they are only useful when the data processed is large enough that the improvements outweigh the extra development time.

2.5. Optimizing Code Predictability

There is much less work to be done with the source code. The compiler knows the proper ordering and optimization rules and will apply them automatically. But any error in prediction for the instructions will cause many more delays than for the data because instructions need to be decoded before they are used by the CPU. Therefore, if possible the code should limit prediction errors.

Branching code should be avoided in the default/normal execution paths. When the expected conditions are met in the code, it should not cause a branch or jump. When a condition has a most likely value, the most likely case code should follow as it will be the one pre-loaded. This happens, for example, when checking for errors or incorrect parameters. We can expect that things work properly and the data provided is valid most of the time. In that case, normal operation should follow the condition and the error handling code can be located further away.

There is another optimization that will happen in the CPU called "Out Of Order execution" (OOO). This is when the CPU detects that two instructions are unrelated and the second execution can be started first to save processing time. This normally does not impact performance unless the second instruction will use a resource (such as reading from the main memory) that the first instruction will also need, thus notably delaying the execution of the first instruction. Memory management instructions of the CPU can prevent OOO from happening if it causes a problem, but once again this will require a precise understanding of the hardware and how memory management instructions work.

2.6. Serialized Code

There are many pieces of an application, including libraries used, that are called by multiple threads running on different cores. There is no way to avoid serialized code completely without duplicating most of the application and OS code which would hurt performances even more.

To avoid concurrent access to serialized parts of the code, the OS uses an internal mutex called a spinlock. A spinlock allows a thread to wait for the needed resource without releasing the core to another thread. Spinlocks are generally only used for very short waiting periods, much shorter than the scheduler's timer period.

Recent high-end processors have added a new feature to reduce the impact of serialized code: transactional registers. Operations done in transactional registers are not applied to the normal registers immediately. This means that instead of waiting for the resource using a spinlock, the second thread performs the calculations in the transactional registers and applies them when the resource is released only if the registers read were not modified in the meantime. In over 90% of cases, the read registers will not have been modified and the second thread will have run without waiting.

3. Practical Issues

3.1. Performance Decrease in Multicore

ISSUE

A real-time application was too slow or caused the CPU load to be close to 100% so an extra RTX core was added to increase performance. But performance did not improve with the extra core or even decreased.

CAUSE

This is most likely due to data synchronization delays. If the application was not developed for a multi-core environment and many variables are shared by the threads on different cores, they will render caches useless and performance will likely decrease due to constant main memory access or cache coherency mechanisms.

POSSIBLE SOLUTIONS

The best solution is most likely to modify the application following the guidelines in section 2 of this paper.

To improve performance without modifying the software, the approach of adding cores is not workable with this specific application. Increasing the frequency of the single core used and possibly the frequency of the RAM would be more efficient.

3.2. Separating Thread Variables Did Not Improve Performance

ISSUE

The real-time application was modified to run on multiple cores. The threads on each core only share a limited number of variables and yet these modifications do not seem to have improved performance significantly.

CAUSE

If separating the variables used by different cores did not improve performance, there is probably false sharing happening. As explained in section 2.3, you need to make sure that variables that are not shared are not on the same cache line as variables which are shared. Variables are not loaded individually by the cache, they are loaded as cache lines of 128 bytes.

POSSIBLE SOLUTIONS

Review the declaration of the variables as explained in section 2.3. Group the different sort of variables in different structures that take whole cache lines to make sure they are separated in the memory and will not interfere.

3.3. Processes on Different RTX64 Cores Interfere

ISSUE

Two independent applications have been set up to run on two different RTX cores, but when one of them has heavy calculations it causes latency in the other.

CAUSE

Two different cores still share resources even if everything has been done to separate them. Even if the I/O used are on separate PCI-e lines and there is no memory shared, the last level cache (LLC) of the CPU is still used by both applications and so is the front side bus (FSB) to access the I/O and RAM. There can be contention in either the LLC or the FSB or both.

When an application does heavy calculations, it likely also uses a large amount of data. This data will pollute the LLC and force the other application to request the data from the RAM again. Loading this data will also create a lot of traffic on the FSB which might get congested. If the other application also needs to access RAM data, these accesses will be longer.

POSSIBLE SOLUTIONS

A first solution would be to limit how fast calculations are performed on the first application to prevent it from polluting the cache and overloading the FSB. Another solution is to get a CPU with more cache and a RAM module with higher frequency to reduce the impact of the heavy calculations.

On a few very high-end processors, Intel has developed a technology called “Cache Allocation Technology” (CAT) which reserves cache space for a specific processor. Intel also announced a technology called “Memory Bus Allocation” (MBA) which reserves an FSB bandwidth for a specific core. This technology is only currently available on the latest series of high-end processors. It is supported by RTX64 3.4.

3.4. More Windows Cores Cause Higher Latency

ISSUE

An RTX application was deployed on a new CPU which is faster and has more cores. The new cores have been assigned to Windows. There has been no change to either the Windows or RTX side of the system and yet there are now delays in RTX applications.

CAUSE

This is caused by bus contention. The front side bus (FSB) which connects the CPU cores to the RAM is a limited resource much slower than CPU cores. It is shared by all cores and any single core can load it completely. There is most likely a Windows application in the system that consumes a lot of data. This application accesses the data through the FSB as fast as possible using all cores available to Windows. With more Windows cores the ratio of FSB bandwidth given to the RTX cores has been reduced, increasing the delays to access the main memory.

POSSIBLE SOLUTIONS

The ideal solution would be to move to a NUMA platform or reserve bandwidth for RTX. But changing the platform to NUMA would require major modifications to the applications and NUMA is not supported by RTX yet. Intel has announced a technology to allocate bandwidth called "Memory Bandwidth Allocation" (MBA), which is supported by RTX64 3.4.

The currently available solutions are to limit the number of system cores to avoid the contention (contention becomes very important when there are more than four cores) or to increase the FSB bandwidth. If possible, our recommendation is to purchase faster RAM and ensure that the Chipset supports this faster frequency.

RTX64 3.4 Support for CAT and MBA

Intel has added features to its newer high-end processors such as cache allocation technology (CAT), memory bus allocation (MBA) and transactional registers to reduce the performance impact and protect critical threads against interferences. RTX64 3.4 now supports these new features by linking them to the priority and affinity settings of real-time threads to enable developers to use newer processors without needing to make changes to their program.

Conclusion

A real-time operating system allows developers to write real-time applications the same way they write Windows applications and handles scheduling and separation of resources with Windows. But some of the resources that are still shared, like the processor cache and front-side bus bandwidth, become bottlenecks of the system in recent processors. This creates performance issues that often seem inexplicable or counterintuitive and impacts the scalability of the system as a whole.

In this paper, we examined what causes these performance issues, how hardware choices can improve performance, and workarounds to help resolve the issues. We also introduced new technology, Intel's CAT and MBA, that further improves performance and is now supported by RTX64 3.4. With this information and these techniques, you can optimize applications for multicore systems and improve scalability for improved results across the organization.

For more information about how to solve performance challenges or the latest features supported by RTX64, please contact your IntervalZero rep or get in touch.

Selecting a Hardware Platform

Follow these guidelines to determine which components will impact your system the most.

a. Processor

What a real-time system requires the most is stability, not burst performance or power saving. Atom processors provide good performance while mobile processors will never be very stable. Features like hyperthreading, boost and sleep states should be disabled if available.

Independent RTX applications should use separate cores and have separate cores from Windows. But to avoid contention issues, we recommend limiting the number of cores to four or being sure that no Windows applications will have bursts of data consumption.

The size and distribution of the cache can often have more impact than the CPU frequency. Generally, the bigger the cache, the faster the code will be executed, although as cache size increases, access latency tends to increase as well. If the cache size is still smaller than the dataset size and many RAM accesses are used, larger cache will reduce overall delays. If the cache size is already larger than the dataset size, then increasing it is counterproductive.

If you have an RTX application that uses multiple cores, having a level 2 cache shared only by the RTX cores may be helpful (see section 1.3).

b. Chipset & RAM

With a heavy application, the RAM access will be the bottleneck. The size of the RAM is not the issue. You need to make sure all the data used by the applications can fit in it; any extra space will not change anything. The frequency of the RAM & Chipset bus, however, is critical and will determine how fast the CPU can access all the data used. For this reason, you should consider as fast a frequency as you can afford.

c. I/O Devices

Any possible bottleneck and conversion delay should be avoided as there will be enough limitations from the RAM access side.

Any device linked to RTX should have its own PCI-e line to avoid delays. New processors only support PCI-e so PCI devices are actually grouped and linked to PCI-e lines by hardware chips. This should be avoided if possible as this extra chip will introduce uncontrolled delays. Only devices directly connected to the chipset will deliver good performance.

All sleep and power saving options should be disabled for the relevant PCI-e lines. These options are in the BIOS and usually now modifiable by the user, so a correctly configured BIOS should be requested from the computer vendor.