

Finding and Fixing Multicore Performance Bottlenecks in HPC Applications



Author's Profile

Erik Hagersten is professor in computer architecture at Uppsala University in Sweden and CTO of Acumem AB. He started his research on parallel architecture at MIT in the early 80s. In the late 80s he was research leader for the architecture group at SICS performing instrumental research on Cache-Only Memory Architectures (and coining its brain-dead acronym COMA). Erik was the chief architect for high-end servers at Sun Microsystems (the former Thinking Machines development team) in the 90s. He has led a research group at Uppsala since 1999, developing the core technology behind Acumem. Erik holds about 100 patents and is a member of the Royal Swedish Academy of Science and Engineering (IVA).

Erik can be contacted at: erik@acumem.com

Multicore processors often run far below expected performance.

Finding the cause of the problem is often a lot harder than fixing it.

It is a complex and challenging task to take full advantage of today's high performance computer architectures. Their deep memory hierarchies make it difficult to reach full performance potential, even for the most astute application programmer. Due to the wide and ever increasing memory gap, today's fast processors often devote more than half of their time waiting for data to arrive from memory or are stalled due to a congested memory bus.

Multicore processor designs can make a bad situation worse (see *Figure 1 in the Application Example at the end of this white paper*), since there is usually less cache per core, resulting in more traffic to the slow memory. Since the cores have to fight for the bandwidth of the memory interface, this can easily become the new bottleneck of the system and limit the performance gains from the use of multiple cores.

The Ideal Performance Analysis Tool

Clearly, developers would want to optimize their application code to address these issues – the question is, how? The ideal tool would first of all analyze the target code to identify the magnitude of each performance problem and its location in the code, rank them all in order of severity, and then provide guidance for the user on how the code can be modified to address each issue.

This seems like a simple problem – rather like finding a needle in a haystack. Or rather, several needles. Oh, and we don't know in advance how many needles are hidden. And no one has told us how big they are... So maybe not so simple, after all. OK, but with a sufficiently powerful performance analysis tool, we could track them all down, right?

The problem we find, surprisingly, is that current performance analysis tools deliver a great volume of information, but provide little or no analysis. In fact, they do not seem to do a great deal to help with performance optimization either, at least not without major expenditure of time and effort on the part of the developer of such data-intensive HPC applications.

Even then, the process requires as much black art as engineering skill. And perhaps worst of all, it is quite possible to spend a great deal of time trying to identify and fix problems without any guarantee at the outset that there will be any significant performance boost to an application even when the process is completed.

A novel approach that provides a faster and much more productive environment for performance analysis and optimization has been developed by Acumem. Based on a unique “fingerprint analysis” technology, it allows the developer to directly locate, quantify and rank performance bottlenecks in a target application, and shows how the code and/or data structures can be modified to address these bottlenecks.

Before we explain how this new technology works, we should first look briefly at the issues with the current approach to performance analysis, to understand what an ideal tool would look like and why the existing technology is incapable of addressing these requirements.

The Classic Approach - Hardware Counters

Currently, most performance analysis tools are based on *hardware counters* counting specific events taking place inside the processor. These hardware counters can, for example, count the number of cache accesses and the number of cache misses for each cache level.

Based on such measurements, properties such as cache *miss ratio* can be computed, representing the likelihood that a requested data is not found in the cache. Some more advanced tools can more or less accurately map miss ratios to the corresponding source code line – a so-called *hotspot* - which might indicate, for example, that a particular source code line is responsible for 35% of all cache misses. However, these tools simply collect cache miss information from the hardware counters and dump them into the lap of the programmer. They give you the haystack – but you still have to find the needles.

The source code line identified by the hotspot typically contains several memory accesses and it is not obvious which one is to blame, nor is it obvious what modification to the code, if any, would give you a performance improvement. Also, understanding the necessary code changes is often non-trivial. Furthermore, no information is provided on what the potential performance gain would be, if any, given such a modification.

The Need for Bandwidth Usage Analysis

All existing hardware counter/timer based tools focus on cache misses. Unfortunately, this completely ignores the cause of an additional performance problem that is particularly characteristic of the multicore system - the data bandwidth issue. In a data-intensive application, the amount of data read and written to DRAM by an application will most often dictate its performance. Just focusing on cache misses will not necessarily identify the accesses responsible for hogging the memory bandwidth.

Most current high-end microprocessors rely on hardware prefetching techniques for avoiding cache misses. The autonomous hardware prefetch logic monitors the memory access patterns and anticipates what data will be requested in the near future and brings them to the cache

prior to their usage. If the hardware prefetcher successfully brings data from the memory to the cache prior to its usage, a future potential cache miss can be avoided and a costly CPU stall due to slow memory access will be eliminated. Even if the prefetch activity successfully removes many of the stalls, they may still consume a large fraction of the available memory bandwidth.

A tool which only reports cache misses cannot record and report the location in the code triggering this prefetch activity, i.e., the source code lines responsible for consuming this precious memory bandwidth. In many programs, more than half of the memory bandwidth is consumed by such prefetch operations. The places in the source code responsible for triggering such fetches should, of course, be put under scrutiny and analyzed as carefully as the instructions causing cache misses, but a tool based on hardware cache miss counters will again fail to provide any useful information. In this case, not only are the tools unable to find the needle, the developer is not even told which haystack to search in.

Time for a New Approach

From the above analysis, it should now be fairly clear what the ideal tool would look like.

First of all, we must acquire a much greater range of cache performance data than is presently available through existing techniques, notably the data required to identify and locate various types of inefficient cache usage. It should also be able to gather comprehensive data on memory bandwidth use. Next, the tool should perform the “heavy-lifting” required to fully analyse all the available data, and present the user with a high-level view of the size, nature and location of the performance bottlenecks in the developer’s application code.

Lastly, having identified each problem, it should provide guidance on how the code can be modified to address the particular issue causing the performance hit.. This combination of features would obviously deliver a much more productive environment for the development of data-intensive applications than any currently available tools.

The most difficult part of this ideal specification is the first part – a completely new approach to gathering and analysing cache performance data from a running application is required.

Enter: Application Fingerprint Analysis

The novel fingerprint analysis technology developed by Acumem fulfils all of these requirements. First of all, an “application fingerprint” is gathered from normal execution on a host computer at runtime. The collected information is very sparse, but contains rich dynamic insights. The fingerprinting does not rely on hardware counters, since the information they offer is simply too low level. The collected information contains dynamic temporal and spatial information about the past and the future of the operation as well as the data it is accessing, such as: When will this data be accessed again and by whom? What is the loop structure around these operations? In what context were the operations called? Is the hardware prefetcher working optimally?

The fingerprint collection is performed on unmodified binaries in a language-agnostic way and works with code produced by literally any compiler. Since the information is collected sparsely, the execution overhead can be kept at a minimum, despite collecting this rich information. Fingerprints from single-threaded as well as multithreaded execution are collected in a completely transparent way.

Since the efficiency of the fingerprint capture technique is very high, this allows for real-sized input datasets, which is essential for studying real-world applications. This is vital if we want to draw any conclusions about modern memory systems and the effectiveness of its caches.

The fingerprint is then analyzed off-line using new and very fast statistical methods. The speed is again important here since we want to answer many “what if” questions and analyze many different alternatives in order to get more insight into the dynamic behaviour.

A More Productive Environment for Performance Analysis

This novel approach, based on sparse fingerprints and statistical analysis, enables us to provide much richer and precise feedback to the user than the more limiting and traditional hardware-counter approach. We can now not only identify the exact operations that cause misses in a cache system or those instructions that waste memory bandwidth, we can also tell the reason for the misses and suggest possible fixes.

Another big plus is that the developer is also able to estimate the potential performance gain for each individual bottleneck identified in the code. So instead of being told: *“There is a “Hotspot” at this point in the code – now go figure!”*, the developer is presented with something much more useful – A **“Slowspot”**.

A Slowspot identifies a piece of code in a specific location with a quantified performance problem that can be improved in a way that would result in a significant performance gain. This new tool based on Acumem’s novel technology (called, rather surprisingly, Acumem SlowSpotter™) provides guidance on how the Slowspots can be rewritten to improve performance. It also shows indexes and efficiency data which helps the developer to work on the Slowspots in the most optimal order, as well as to measure the level of actual improvement from code changes as they are made. Last, but not least, the tool is fast enough to provide a short turn-around time based on real-sized input data.

Finally, it is easy to overlook one of the most powerful features of such a tool – the ability to inform the application developer quickly and simply whether a speed improvement can be achieved in a piece of code – **or not**. The “or not” option is a key issue in a productive environment for performance optimization, since if little or no improvement is possible, it eliminates the need for any further expenditure of the developer’s valuable time. In any case, since the Slowspots can be ranked in order of performance to be gained, the developer can choose the optimum cut-off point, below which the prospective speed increase is outweighed by the additional programming effort required.

Conclusion

Historically, performance analysis tools have provided large volumes of data, but leave a great deal of the required analysis to the developer. This makes the process of performance analysis and optimization very laborious. Nor do they typically have any predictive capability - the estimation of performance gain is left to the user.

At the same time, the problem of lost performance is becoming even worse with the shift to multicore processors, where low scalability due to poor cache and memory bandwidth utilization is commonly experienced, particularly in data-intensive applications.

Acumem's fingerprinting technology provides a fundamentally different approach to the problem. Firstly, an unprecedented level of detailed information is provided on application performance. Secondly, the detailed analysis is done by the Acumem Slowspotter™ tool, rather than by the developer, providing the location and nature of specific performance Slowspots in the code. Lastly, the tool provides detailed advice on how each Slowspot can be fixed.

And if all this seems too good to be true – then you can try it out for yourself on your own code free of charge! Acumem SpotLite™ is a version of Acumem SlowSpotter™ available for free download from www.acumem.com. This allows potential users to evaluate the effectiveness of the tool for themselves. Although offering more limited functionality, Acumem SpotLite™ uses the same underlying technology as Acumem SlowSpotter™ to analyse performance bottlenecks in the user's application code.

About Acumem

Acumem is a provider of intelligent software which analyzes and optimizes application software performance in single- and multi-core environments. Acumem contributes to their customers' success in maximizing the benefits of multi-core technology, achieving the full potential of their systems. Acumem works closely with leading companies in the multi-core market such as HP, Sun and AMD. Acumem is a privately held company based in Uppsala, Sweden. For more information, visit www.acumem.com

Application example follows . . .

Application Example – Multicore and SPEC CPU 2000 (470.lbm)

It is quite easy to assess how much the system throughput really improves if you utilize all four cores of a quad core system compared with a single core. This also provides a useful demonstration of the effectiveness of the Acumem fingerprint technology, since we can show the performance boost from restoring lost memory bandwidth by a simple “before & after” test.

Actually, all the needed experimental data are readily available on the web. The SPEC organization is chartered to collect a representative set of applications to be used in comparison between different hardware and software components. The current set of CPU applications is a collection of 29 widely used applications from different areas. **Figure 1** shows how many more operations the floating point applications in the CPU 2006 can achieve if four instances of each application are run on four cores compared with a single application run on one core. This chart is based on data published by Dell, running on a Core2 Quad X5365 3.0GHz, 1333MHz FSB, equipped with 800 MHz DDR2 RAM. Out of the 17 applications, 8 of them do not even see a two-fold throughput improvement.

We selected the benchmark with the worst scalability, 470.lbm, for a test. After code optimization aided by Acumem Slowspotter™, we compared the execution time of the original and modified versions of the application on a Intel Core2 Quad Q6600 2.4GHz, 1066MHz FSB, equipped with 800 MHz DDR2 RAM. The result is shown in **Figure 2** below. Note that the *unoptimized* scalability of 470.lbm is slightly better (1.4 vs. 1.2) than shown in Fig.1 on this different system configuration. As a matter of interest, Slowspotter can also be used to predict such performance variations between cores, using additional capabilities not mentioned in this article.

When 470.lbm is running on all four cores the performance improvement is less than 1.5 times that of a single core. Slowspotter™ revealed the severe impact of the competition for the shared caches and memory bandwidth when running on all four cores, and indicated how these issues could be addressed. This more than doubled the speed of the benchmark, from 1.4 to 3.2 times relative to the single core performance. This shows not only the effectiveness of the Slowspotter™ tool, but also the potential performance loss due to poor memory bandwidth utilization in multicore applications.

Figure 1: SPEC2006 FP throughput improvements on 4 cores

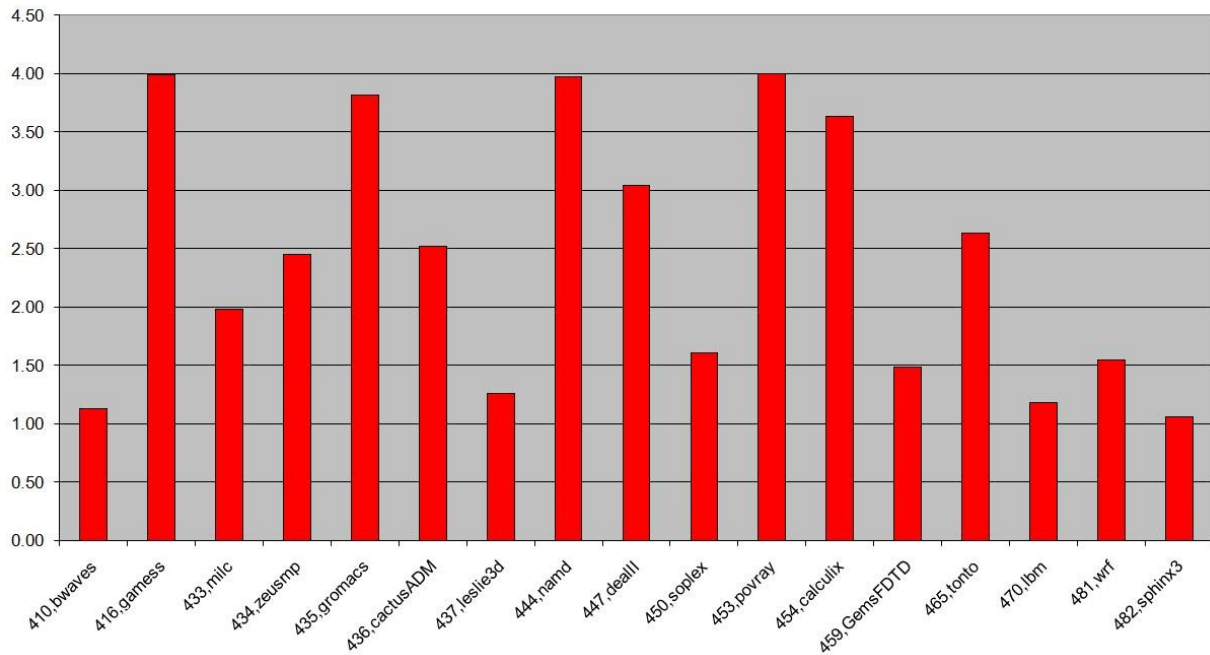


Figure 2: 470.lbm code, before and after SlowSpotter analysis & optimization

