# Virtual Aggregated Processor in Multi-core Computers

Z. Huang† A. Trotman† J. Zhang†, X. Jia† M. Nowostawski‡ N. Rountree† P. Werstein†
†Department of Computer Science
University of Otago, Dunedin, New Zealand
Email:{hzy;andrew;fei;rountree;werstein}@cs.otago.ac.nz
‡Department of Information Science
University of Otago, Dunedin, New Zealand
Email: mariusz@nowostawski.org

## Abstract

*Parallel computing has been in the spotlight with the advent of multi-core computers. The popular multithreading model does not scale very well when there are hundreds or thousands of cores, since it can only help exploit coarse-grained parallelism. There exist a lot of fine-grained parallelism to be exploited in I/O tasks and memory accesses during execution of a thread. Our Counter-Amdahl's Law tells us that it is more effective to parallelize the serial fraction of a parallel algorithm rather than the parallelized fraction in order to maximize the speedup. In this paper, we have proposed a Virtual Aggregated Processor that is aiming at speeding up execution of a thread through exploiting the fine-grained parallelism in I/O tasks and memory accesses. We have proposed and implemented two techniques, helper thread and I/O specialization, to demonstrate the potential effectiveness of the Virtual Aggregated Processor technology.*

**Key Words:** Multi-core, Virtual Aggregated Processor, Counter-Amdahl's Law, Helper Thread, I/O specialization

## 1 Introduction

Computer architectures and the computer industry are being transformed by the advent of multi-core (MC) technology. According to Moore's Law [1, 25], the number of transistors in a single silicon die (also known as a chip) doubles every two years, so the best evidence is that the core count will continue to grow. At present, the most advanced general-purpose processor available, Sun Microsystems Ultra-SPARC T2 [8], features eight processing cores capable of running 64 independent threads (processes) on a single chip, but future chips, including those from other manufacturers like Intel and AMD, will undoubtedly involve hundreds of simultaneously operating cores.

MC technology is a disruptive technology because it offers massive increases in processing capacity on a single computer. With this new technology, server farms (such as those operated by Google, Yahoo!, and others) will consume a fraction of the power and be far less costly to operate. This technology might make server farms redundant due to high-level integration of processing units on silicon dies.

MC technology also opens new opportunities for system- and application-level software to harness the power of multiple cores. However, to realise these opportunities, fundamental research issues need to be addressed [11]: how can the application-level software utilise efficiently the available processing power? How can the system-level software better support the parallelization of applications?

Current parallelization software such as OpenMP [2] and MPI [3] are all based on multithreading technology, which enables multiple threads (or processes) to work in parallel on the same computational problem. However, for most application software, the number of threads that can be adopted is limited due to Amdahl's Law [4, 10]. Though we may use one core for each thread in the multithreading model, we may still waste the power of hundreds of cores that are idle. The research question for us is "Can we use the idle cores to speed up a thread already running on a core?"

Another problem with the multithreading model is that it can only help exploit coarse-grained parallelism at the thread level. Much potential fine-grained parallelism, such as I/O tasks, is hidden from threads. The research question for us is "Can we exploit the fine-grained parallelism in the sequential execution of a thread?"

To address the above research questions, we have proposed a Virtual Aggregated Processor (VAP) project to examine opportunities for fine-grained, low-level parallelism within individual applications and threads. VAP is underpinned by two important computer science principles: specialization is more effective than generalisation, and Amdahl's Law bounds the effect of parallelism; and is sup-

ported by two important trends: the breakup of monolithic operating systems into discrete services/devices, and the re-emergence of virtualisation. We leverage these principles and trends in a novel manner to increase the number of tasks that can be parallelized in a single thread and implement these tasks in virtual devices. This will maximise the use of available cores, thus improving processing capacity and reducing energy consumption through more effective utilisation.

This paper will introduce our idea and principles behind VAP and its preliminary research results. The rest of this paper is organized as follows. Section 2 introduces our Counter-Amdahl's Law in order to emphasize the importance of parallelizing serial fraction of a parallel algorithm. In Section 3, we propose the VAP architecture based on existing hypervisor technology such as Xen [6, 13]. In Section 4, we present the implementation of a helper thread on MC computers and demonstrate its preliminary performance. Section 5 presents our I/O specialization techniques to improve I/O tasks and demonstrate its effectiveness for special domains such as information retrieval. Finally, our future work is suggested in Section 6.

# 2 Counter-Amdahl's Law

The efficiency of a parallel algorithm is normally measured with its speedup relative to its sequential version. Speedup is formally calculated with the following formula.

**Definition 1** *Speedup of a parallel algorithm*
The speedup of a parallel algorithm is

$$S = \frac{T_s}{T_p}$$

where $T_p$ is the execution time of the parallel algorithm, while $T_s$ is the execution time of its sequential version.

Suppose $f$ is the fraction of the computation that is serial in a parallel algorithm, and $p$ is the speedup for the parallel fraction of the algorithm. Based on the above formula, the overall speedup of the algorithm can be expressed as:

$$S = \frac{T_s}{fT_s + \frac{(1-f)T_s}{p}} = \frac{p}{1+(p-1)f}$$

In 1967, computer architect Gene Amdahl argued that the maximum speedup of a parallel algorithm is bounded by the serial part of the algorithm [4, 10]. More formally, the Amdahl's Law is described below.

**Definition 2** *Amdahl's Law*
Suppose $f$ is the fraction of the computation that cannot be parallelized in a parallel algorithm, and $n$ is the number of processors working on the concurrent parts of the algorithm. The speedup of the algorithm is bounded by

$$S(n) = \frac{n}{1+(n-1)f}$$

Even with an infinite number of processors (i.e. $n \rightarrow \infty$), the maximum speedup is bounded by

$$S(n)_{n\rightarrow\infty} = \frac{1}{f}$$

With Amdahl's Law, we know that the serial fraction of a parallel algorithm has a significant impact on the scalability of the algorithm. If that fraction is relatively large, there will be no speed increase for the algorithm with a given problem size on a small number of processors. For example, suppose $f$ is 0.5 in an algorithm, no matter how many processors are used, the maximum speedup of the algorithm is no more than 2.

To reduce the effect of Amdahl's Law, we should reduce the serial fraction of a parallel algorithm. To emphasize the importance of parallelizing the serial fraction of an algorithm, we propose the following Counter-Amdahl's Law.

**Definition 3** *Counter-Amdahl's Law*
Suppose $f$ is the fraction of the computation that is serial in a parallel algorithm, and $p$ is the speedup for the parallel fraction of the algorithm. Then, if $p > (1-f)/f$, which means $p$ is greater than the ratio between the parallel fraction and the serial fraction, it is more efficient to improve the serial fraction rather than the parallel fraction in order to increase the overall speedup of the algorithm.

**Proof:** Suppose an additional number of processors is used to improve either the serial fraction or the parallel fraction, which is accelerated by $m$ times. Then the overall speedup for the algorithm with the improvement on the serial fraction is

$$S(s) = \frac{T_s}{\frac{fT_s}{m} + \frac{(1-f)T_s}{p}} = \frac{pm}{pf+(1-f)m}$$

and the overall speedup for the algorithm with the improvement on the parallel fraction is

$$S(p) = \frac{T_s}{fT_s + \frac{(1-f)T_s}{pm}} = \frac{pm}{pmf+(1-f)}$$

Since $p > (1-f)/f$, then we have $pf + f > 1$; then multiply $m-1$ on both sides, we have $pf(m-1)+f(m-1) > m-1$; then unravel the left side, we have $pfm - pf + mf - f > m-1$; then shift around the terms, we have $pmf + (1-f) > pf + (1-f)m$; then inverse both sides, we have $1/(pf+(1-f)m) > 1/(pmf+(1-f))$; and finally multiply $pm$ on both sides, we have

$$\frac{pm}{pf+(1-f)m} > \frac{pm}{pmf+(1-f)}$$

Therefore, $S(s) > S(p)$.

From the Counter-Amdahl's Law, we know that when $p$ becomes large, it is more efficient and more effective to improve the serial fraction rather than the parallel fraction. When $p \to \infty$, $S(s)_{p\to\infty} = 1/mf$, which means the overall speedup of the algorithm can be improved about $m$ times if the serial fraction is speeded up by $m$ times.

For example, suppose $f$ is $0.1$ and $p$ is $100$ in an algorithm. If its serial fraction is accelerated by 2 times ($m = 2$), then the new overall speedup of the algorithm is approximately 16.94, which is about 1.8 times of its original speedup 9.17. This means, in this situation, if we can use two or three additional processors to accelerate the serial fraction by 2 times, its overall effect is much better than to use 100 additional processors to improve the parallel fraction by 2 times.

# 3 Virtual Aggregated Processor (VAP)

Current parallelization software [2, 3] is based on multithreading technology, which enables multiple threads to work in parallel across multiple processors. However, the ability of multi-threading techniques to parallelize tasks is limited because multi-threading is coarse-grained, and the OS-level and architecture-level parallelism cannot be exploited through multi-threading. As stated in the Counter-Amdahl's Law, it is more efficient to improve the serial fraction of an algorithm. In a sequential execution of a thread, there is a lot of fine-grained parallelism to be exploited at the OS and architectural level. As far as we know, there are two approaches to parallelizing an execution of a thread. One is to parallelize the system calls and libraries to be invoked by a thread. The other is to parallelize memory and I/O accesses. The VAP project is taking both approaches to achieve maximum possible parallelism for multi-threading applications, in order to counter the effect of Amdahl's Law.

Our approach for VAP is based on the important principle that specialization, rather than generalization, is the most effective approach to high performance computing. A prominent advocate of this belief is Gordon Bell, a veteran in parallel computing from Digital Equipment Corporation and now a Microsoft evangelist, who confirmed that specialization was the most effective approach in the history of high performance computing [14]. Therefore, we are investigating specialized parallel algorithms and techniques based on the specialties of both the specific VAP applications and the specific multi-core architectures.

VAP is a virtual processor comprised of multiple cores coordinated by software. It can be used as a reconfigurable, software-powered, specialized virtual device that can potentially contain dozens of cores, as shown in Figure 1. The novelty of VAP is that it enables multiple cores (or processors) to facilitate the execution of a single thread, while traditional operating system software focuses on how multiple threads best share a single processor.

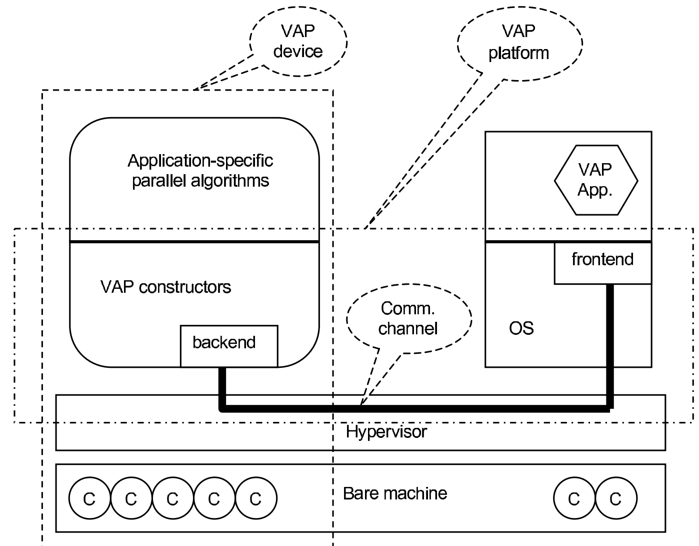


Figure 1: VAP architecture

To develop VAP virtual devices, we will leverage another transformation in computing: virtualisation. While not itself a new idea, virtualisation technology recently developed by companies such as XenSource [6, 13] and VMware [5] can enable multiple virtual machines to run on a single physical computer. All major chip manufacturers are supporting virtualisation in computer hardware and this gives us the opportunity to develop efficient specialized virtual devices, separate from OSs, via the chips hardware support.

As shown in Figure 1, a VAP is positioned on a thin layer of virtualization software called the hypervisor, which controls access to and partitions the physical resources of the bare machine. An application in a standard virtual machine can access the power of a VAP via a virtual device driver called the frontend. The frontend interacts with a VAP via a communication channel connecting to a backend in the VAP. The VAP software consists of VAP constructors such as the helper thread and pipeline, which are the basic building blocks of VAP, and application-specific parallel algorithms. We will develop VAP prototypes in information retrieval and data mining as examples of the VAP approach. There may be different types of VAPs or different instances of the same VAP inside a multi-core computer. Based on the VAP platform, which consists of the VAP constructors, the communication channel, the frontend, and the backend, other high performance VAP products in domains such as packet classification, network intrusion detection, and Internet traffic analysis can be investigated and developed.

The VAP technology will provide both high performance and flexibility to computer operating systems. The high performance will result from the parallel techniques specialized for the application domain and the multi-core architecture, while the flexibility comes from the virtualisation technology supporting VAPs, which does not require modifications of existing operating systems. In addition, the VAP approach is cost effective and easy to operate compared with hardware devices such as specialized network processors, since a VAP is just a software system running on multiple cores and can be deployed or removed with a few keystrokes.

To prove the concept of VAP, we have implemented two VAP-related techniques: helper thread and I/O specialization. The helper thread is a constructor used to parallelize the memory accesses, while I/O specialization is a technique that can take advantage of special domain knowledge in information retrieval to improve I/O tasks such as disk I/O. These techniques are described in details in the following sections.

# 4 Helper thread

We have implemented the idea of helper thread in our parallel programming environment called Maotai [29] for multi-core computers. Maotai has implemented our novel View-Oriented Parallel Programming (VOPP) [19, 20] on Sun Microsystems UltraSPARC T1 (aka Niagara) [7]. The novelty of VOPP is that it divides shared memory into views in order to help remove data races and reduce communication overhead in parallel programs. For more details of VOPP, the reader is referred to [19].

## 4.1 View prefetching

VOPP provides sufficient information for view prefetching in VOPP, because view primitives like *acquire_view* and *release_view* have to be used when a view is accessed. Those view primitives, along with view information, can inform the system to prefetch view data from memory to caches.

Previous work on prefetching techniques [15, 17, 18, 21–24, 26] have received much attention recently with the advent of chip-level multithreading technology. To prefetch data accurately and efficiently, efforts have been put into region selection, which identifies the appropriate regions to include a piece of helper code, and phase detection which identifies the right timing to run the helper code [23].

However, without the help of the above techniques, VOPP can provide the right information for both the prefetching regions and the prefetching timings. When a view is acquired, it is almost for sure that the memory space of the view is about to be accessed due to the view-oriented feature of VOPP. When a view is created with the *alloc_view* primitive, the address and the length of the memory space of the view are recorded. With this information, we can prefetch the memory space of a view at the view acquiring time.

For our view prefetching, we have tried a few different ways. At first, we used the PREFETCH instruction provided by UltraSPARC T1, but it cannot help load a large view into a cache in time. We then used an alternative solution–helper thread, which does prefetching as a separate thread. Helper threaded prefetching is a technique which proved to be promising on multi-core and hyperthreading platforms [21, 22, 24].

With the help of the view information discussed above, our helper thread can adapt to the dynamic behavior of a running application. That means, it works effectively despite a different input data set each time an application is given. The communication between the helper thread and the task thread (aka the helped thread) is achieved by a shared variable that contains the identifier of the view being acquired. In our implementation, we make the helper sleep in a wait queue initially. When a view is being acquired, the task thread wakes up the helper, which then checks the shared variable to find out which view should be prefetched.

In previous research work, helper threaded prefetching is used in both chip-level multiprocessors (CMP), which have multiple cores inside one chip, and Simultaneous Multi-Threading (SMT) [28] processors, which physically support simultaneous threads in a single core. The implementation of a helper with a hardware thread inside an SMT processor is called a tightly-coupled helper, and the helper implemented with another core in a CMP is called loosely-coupled. It had been suggested that a tightly-coupled helper incurs contention of the same core that is shared among multiple threads [21, 24]. However, a helper thread that is located in the same core as the task thread can actually help prefetch the data into the L1 cache which is much closer to the CPU than the L2 cache. Although the difference of the speed between the L1 cache and the L2 cache is not as significant as that between the L2 cache and the memory, there are chances that tightly-coupled helpers would provide further performance gain when we perform read-only access (e.g. *acquire_Rview*). Previous research work could not compare and evaluate both the loosely-coupled and the tightly-coupled approaches with experimental results. Fortunately, with the CMT technology in UltraSPARC T1, which supports both CMP and SMT, we can now evaluate them on the same architecture.

## 4.2 Performance of the helper thread

To evaluate our preliminary implementation of helper threads, we divide our experiments into two parts. The first part involves cache misses and the second part involves gen-

eral performance of the helper threads.

The benchmark program is a sum program. It adds all the integers from a shared array. It is selected because it is a memory-intensive program that has a regular memory access pattern, which makes it an ideal program to show the effectiveness of helper threads.

Since Linux dynamically schedules the processes to any physical cores, to perform our test, we have to bind the processes to specific physical cores with the *set_affinity()* system call in Sparc64 Linux.

In order to get accurate profiling of cache misses, the L2 cache and the L1 caches are thoroughly cleared before the computation starts. Since there are no libraries for performance profiling for UltraSPARC T1, we have to access directly the two performance counters, namely PIC and PCR, by calling the *perfctr()* system call in Sparc64 Linux.

Cache misses are shown in Table 1 and 2 for two data set sizes, 4K and 100K, which are used for the integer array in the sum program. The results in the tables are collected for the sum program running with one task thread and one helper thread (if helper threaded prefetching is used).

| *Helper Type* | *task thread* | | | *helper* | |
|---|---|---|---|---|---|
| | *L1* | *L2* | *Ticks* | *L1* | *L2* |
| $MT_{tc}$ | 10 | 1 | 25685 | 245 | 64 |
| $MT_{lc}$ | 252 | 1 | 28484 | | 64 |
| $MT_{non}$ | 252 | 63 | 34362 | | |

Table 1: cache misses, 4K data

| *Helper Type* | *task thread* | | | *helper* | |
|---|---|---|---|---|---|
| | *L1* | *L2* | *Ticks* | *L1* | *L2* |
| $MT_{tc}$ | 482 | 131 | 510084 | 5875 | 1447 |
| $MT_{lc}$ | 6278 | 1 | 567828 | | 1564 |
| $MT_{non}$ | 6278 | 1563 | 703808 | | |

Table 2: cache misses, 100K data

In the above tables, $MT_{tc}$, $MT_{lc}$, and $MT_{non}$ stand for VOPP task thread with a tightly-coupled helper thread, VOPP task thread with a loosely-coupled helper thread, and VOPP task thread without any helper, respectively. The columns $L1$ and $L2$ are the L1 and L2 cache misses. The column *Ticks* is the number of CPU ticks cost by the task thread.

From Table 1 and 2, we can see that the helper thread can significantly decrease the CPU ticks of the task thread. Compared with $MT_{non}$, the tightly-coupled helper can dramatically decrease both L1 and L2 cache misses by 96% and 98% respectively, while the loosely-coupled helper can decrease only L2 cache misses by 98%.

However, when the data set size is larger, e.g. 100K in Table 2, the count of L2 cache misses for $MT_{tc}$ is higher than that of $MT_{lc}$, and its L1 cache misses is also increasing. This is largely due to the interference between the task thread and the helper thread competing for resources in the same core. Nevertheless, there is a significant performance gain (28%) by the tightly-coupled helper thread according to the CPU ticks, which is attributed to the decrease of L1 and L2 cache misses (92% and 91% respectively).

We can also notice that no matter whether we run the helper thread on the same core or not, the total number of L2 cache misses from both the task thread and the helper thread is larger than that of $MT_{non}$. This also applies to L1 cache misses. The above result is expected due to the interference between the two simultaneous threads.

Since our experimental results have shown that tightly-coupled helper threads can perform better for read accesses, we currently adopt the tightly-coupled approach for the *acquire_Rview* in our implementation of VOPP. However, since L1 cache is write-through in T1, the tightly-coupled helper cannot provide the benefits mentioned above and will incur more contention. Therefore, we use loosely-coupled helpers for write accesses instead. The performance benefit from the reduced cache misses is reflected in the improved performance of the parallelized sum program, which is shown in Figure 2.

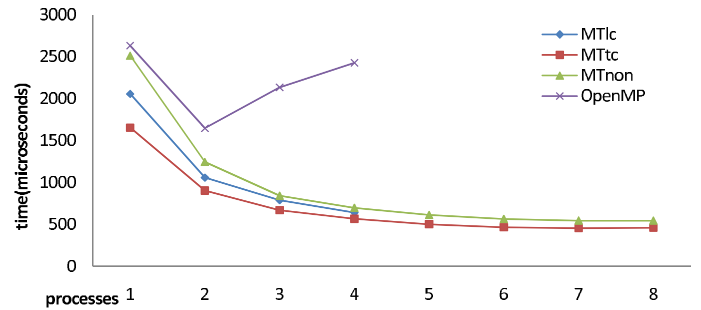


Figure 2: performance of helper threaded prefetching for VOPP

Figure 2 shows the performance of a simple sum program that adds up 25,000 integers randomly selected from an array of 100,000 integers. $MT_{tc}$, $MT_{lc}$, $MT_{non}$ have the same meaning as above. We only use up to 4 cores to perform this test for $VOPP_{lc}$, because there are only 8 cores in the T1 chip and thus we can only have 4 cores for the task threads while the other 4 cores are used by the helper threads. For comparison purposes, we show the time cost of the corresponding OpenMP program, which suffers from performance penalties due to its large synchronization overhead in the program.

For VOPP with helper threaded prefetching, we can see a significant improvement of performance. With tightly-coupled helper threads, VOPP achieves an even better speedup (34% better than $MT_{non}$ at one process). However, when the number of processes is increasing, the performance gap between $MT_{non}$ and $MT_{lc}$ is decreasing. This is expected because when the data loaded into each

process becomes smaller, the helper threads are less effective in terms of cache prefetching. This also applies to the decreased performance gap between $MT_{tc}$ and $MT_{non}$, which is also partially attributed to the high L2 cache misses of $MT_{tc}$ as mentioned above.

Due to limited time, we have not integrated the helper threads in more applications yet. Tests on other benchmark applications with helper threaded prefetching will be carried out in our future work.

# 5 I/O Specialization

General-purpose operating systems usually provide disk I/O optimization in the kernel by providind general-purpose algorithms as they have to serve various kinds of applications. However, for special-purpose applications, it is better to deploy application level optimization. Application level I/O optimization allows applications to define their own data structures and I/O algorithms. In VAP, special-purpose I/O optimization is defined in the backend, rather than directly in the application. In this section, we use search engine as a demonstration of deploying special-purpose I/O optimization algorithms, including caching, prefetching and scheduling.

## 5.1 Special-purpose Algorithms

Caching can be applied to either query results or posting lists. Caching query results not only optimizes disk I/O, but also avoids reprocessing of query evaluation. However, queries tend to have low frequency of repetition and thus result in a high cach miss rate [12].

Caching posting lists, on the other hand, can achive a high hit ratio [12]. Caching policies, like LRU or Least Frequently Used (LFU), define how efficiently a finite amount of cache is used for large amounts of data on disk. This is so called dynamic, due to frequent update of cache memory. One of the challenges for dynamic caching is that caching variable sized posting lists is quite difficult in terms of efficiency and cache memory management.

Instead of frequent update of cache memory, another way of caching posting lists is to define which posting lists are the most important and then let them stay in cache memory without eviction. This is so called static caching. Because the most important posting lists are already cached, there is no need to further update the cache memory. The question is how to define the importance of postings lists. One solution is to choose query terms which have the highest query-term frequencies $f_q(t)$. Another solution can be based on document-term frequencies $f_d(t)$. Terms with high $f_d(t)$ have long posting lists, thus consuming more cache memory. However, caching long posting lists reduces disk I/O as it takes more time to read long posting lists from disk. Baeze-Yates et al [12] argue that the importance should be defined as $\frac{f_q(t)}{f_d(t)}$ (Posting lists with high query-term frequencies and short length in size are preferred). Static caching simplifies cache management by eliminating cache replacement policy. However, there is a price to pay by doing so. Because the importance of posting lists is based on the analysis of the early query log, the importance needs to be re-defined if incoming queries are outside the coverage of the early query log. The operation of dealing such problem is to re-fill the cache, resulting in very low hit ratio and performance decrease.

Both dynamic and static caching have pros and cons. Dynamic caching is good at keeping up with frequent changes in queries, while static caching simplifies cache management policy and results in high hit ratio under normal circumstances. An obvious question to address is the possibility of combining both dynamic and static caching together. The potential problems are: (1) which posting lists to cache dynamically and which to cache statically, (2) how to distribute cache memory usage among them, (3) is there a performance gain for such a combination.

Prefetching and scheduling are straightforward. The access pattern of a search engine can be predicted by the query terms. Posting lists for the next term can be prefetched while the current term is being processed.

If we consider that posting lists are sorted in terms of the alphabetic order of the dictionary terms, we can define a new scheduler which sorts disk I/O requests in the order of the dictionary terms. The sorting can be either local or global, where local means sorting terms in a single query and global means sorting terms in several queries executed concurrently. Local sorting has quick individual response time while global sorting has better overall performance. One thing to note for the scheduler is that once the sorting is done in ascending order, then next time the sorting should be done in descending order, and so on. This allows the disk head to move from the centre of the disk to the edge and then back to the centre.

| Size of the Collection | $\approx$ 18GB |
|---|---|
| Number of Documents | 1247753 |
| Number of Unique Words | 8849995 |
| Average Document Length | 975 |
| Size of the Postings File | $\approx$ 818MB |
| Size of the Dictionary File | $\approx$ 394MB |
| Size of the Disk Image | $\approx$ 1.2GB |

Table 3: Summary of the TREC-2002 Web Track collection

## 5.2 Performance

We used the document collection of the TREC-2002 Web Track [16], summarised in Table 3, and the TREC 2007 Million Query Track [9] for evaluation of the document col-

| Drive Specification | ST380215A |
| --- | --- |
| Capacity | 80GB |
| Spindle Speed | 7200RPM |
| I/O Data Transfer Rate | 100MB/sec |
| Cache Buffer | 2MB |
| Average Latency | 4.16ms |

Table 4: Specification for the test disk [27]

lection. The disk image had a copy of the postings file at the beginning, followed by the dictionary file starting at a new sector location. The raw disk image was then copied to the disk using the Linux dd command.

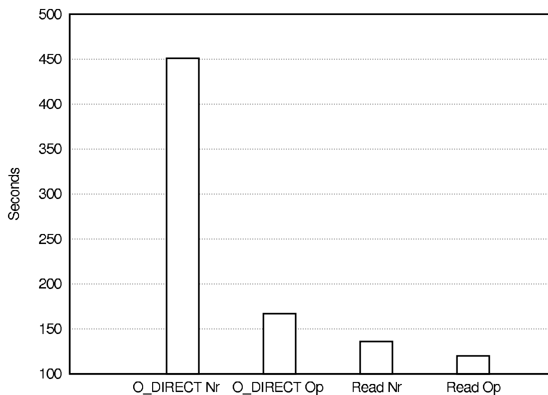


Figure 3: I/O access comparison of search engines (Nr and Op stand for Normal and Optimized respectively)

We conducted tests on a PC with an Intel single core Pentium 4 CPU running at 2.4GHz, with 512KB of L2 cache and a speed of 533MHz for the Front Side Bus. The system has 768MB of DDR266 main memory. We used separate disks for installation of kernels and testing. The testing disk is the IDE primary master, while the kernel disk is configured as the primary slave. The kernel disk is not needed for the performance test. Table 4 shows the specification for the testing disk as stated by the manufacturer. The chosen Linux distribution was Debian Etch, with the default Linux kernel 2.6.8.

The RDTSC instruction was used for timing purposes and the number of cycles returned by RDTSC was converted to seconds by dividing by the CPU frequency. In order to minimise bias in testing results, we re-booted the testing machine for each run and the swap partition used for Linux was disabled. Each test was run five times and the average was taken as the final result.

We carried out four tests: (1) O_DIRECT Normal, (2) O_DIRECT Optimized, (3) read() Normal and (4) read() Optimized. The optimized tests had the application level I/O optimization enabled, while the normal tests had no application level optimization. The O_DIRECT tests was essentially the same as the read() tests, and the only difference was disk access. The O_DIRECT option, when specified for system call read(), bypass the Linux I/O subsystem.

Figure 3 shows the results. The I/O read time was the total time taken reading the postings from the disk. The O_DIRECT Normal test took the longest time for reading I/O (about 451 seconds). The O_DIRECT optimized test performed about 73.4% better than O_DIRECT Normal. The read() Normal and optimized tests took about 167 and 136 seconds, respectively. Interestingly, read() optimized also benefited from the application level optimization. O_DIRECT Optimized beat read() Normal by 47 seconds (28% improvement), showing that application-specific optimization is superior to what is offered by the Linux kernel. O_DIRECT Optimized also beat read() Optimized by 16 seconds (11% improvement), demonstrating the overhead of the Linux I/O subsystem when the application provides its own I/O optimization.

# 6 Conclusions and future work

Based on our Counter-Amdahl's Law, we have proposed the Virtual Aggregated Processor (VAP) project and its architecture. Two VAP-related techniques, helper thread and I/O specialization, have been proposed and implemented. These techniques have demonstrated that there is a promising potential in exploiting fine-grained parallelism and optimizing the I/O tasks in the execution of a thread, as targeted by the VAP project.

In the near future, we will implement the VAP architecture based on the Xen hypervisor on multi-core computers. We will use information retrieval and data mining as example domains to demonstrate the effectiveness of the VAP technology in terms of parallelization and specialization. The communication channel and the interface between a VAP and a VAP application will be proposed, standardized, and implemented for development of VAPs for other domains.

# Acknowledgment

The authors would like to thank Stuart Barson for his excellent comments and suggestions on the VAP project.

# References

[1] http://en.wikipedia.org/wiki/Moore's_law.

[2] http://www.openmp.org.

[3] http://www-unix.mcs.anl.gov/mpi/.

[4] http://en.wikipedia.org/wiki/Amdahl's_law.

[5] VMware. http://www.vmware.com/.

[6] XenSource. http://www.citrixxenserver.com.

[7] UltraSPARC Architecture 2005 specification. http://opensparc-t1.sunsource.net/, 2005.

[8] UltraSPARC T2. http://www.sun.com/processors/UltraSPARC-T2/, 2007.

[9] James Allan, Ben Carterette, Javed Aslam, Virgil Pavlu, Blagovest Dachev, and Evangelos Kanoulas. Million query track 2007 overview. In *Proceedings of TREC*, 2008.

[10] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *In Conference Proceedings of American Federation of Information Processing Societies*, volume 30, pages 483–485, 1967.

[11] K. Asanovic et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, University of California at Berkeley, December 2006.

[12] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 183–190, New York, NY, USA, 2007. ACM.

[13] Barham, P., et al. Xen and the art of virtualization. In *In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[14] Gordon Bell. Paths and cul de sacs on the endless road to supercomputing. In *invited talk at the Symposium on Modern Computing at the 100th Centenary of John Vincent Atanasoff at Iowa State University*, 2003.

[15] T.F. Chen and J.L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transations on Computers*, 44(5):609–623, May 1995.

[16] Nick Craswell and David Hawking. Overview of the trec-2002 web track. In *TREC*, 2002.

[17] G. K. Dorai and D. Yeung. Transparent threads: Resource sharing in SMT processors for high single thread performance. In *In Porceedings of International Conference on Parallel Architectures and Compilation Techniques 2002*, page 30, 2002.

[18] J. Dundas and T. N. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *International Conference on Supercomputing*, pages 68–75, 1997.

[19] Z. Huang and W. Chen. Revisit of View-Oriented Parallel Programming. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 801–810, 2007.

[20] Z. Huang, M. Purvis, and P. Werstein. Performance evaluation of View-Oriented Parallel Programming. In *Proceedings of the 2005 International Conference on Parallel Processing (ICPP05)*, pages 251–258, June 2005.

[21] C. Jung, D. Lim, L. Lee, and Y. Solinhin. Helper thread prefetching for loosely-coupled multiprocessor systems. In *Proceedings of 20th IEEE International Parallel & Distributed Processing Symposium*, 2006.

[22] D. Kim et al. Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, pages 27–38, 2004.

[23] J. Lee, Y. Solihin, and J. Torrellas. Automatically mapping code on an intelligent memory architecture. In *In Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 121–132, 2001.

[24] J. Lu et al. Dynamic helper threaded prefetching on the Sun UltraSPARC CMP processor. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, 2004.

[25] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 1965.

[26] T. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems*, 16(1):55–92, February 1998.

[27] Seagate. Barracuda 7200.10 pata. Product Maunul, Seagate Technology, August 2007.

[28] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *In Proceedings of International Symposium on Computer Architecture*, pages 392–403, 1995.

[29] J. Zhang, Z. Huang, et al. Maotai: View-Oriented Parallel Programming on CMT processors. In *In Proceedings of the 2008 International Conference on Parallel Processing (ICPP08)*, September 2008.