



UPPSALA
UNIVERSITET

*Digital Comprehensive Summaries of Uppsala Dissertations
from the Faculty of Science and Technology 176*

Methods for Creating and Exploiting Data Locality

DAN WALLIN



ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2006

ISSN 1651-6214
ISBN 91-554-6555-2
urn:nbn:se:uu:diva-6837

Dissertation presented at Uppsala University to be publicly examined in Polacksbacken, building 2, room 2446, Uppsala, Wednesday, May 24, 2006 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract

Wallin, D. 2006. *Methods for Creating and Exploiting Data Locality*. Acta Universitatis Upsaliensis. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 176. 37 pp. Uppsala. ISBN 91-554-6555-2.

The gap between processor speed and memory latency has led to the use of caches in the memory systems of modern computers. Programs must use the caches efficiently and exploit data locality for maximum performance. Multiprocessors, built from many processing units, are becoming commonplace not only in large servers but also in smaller systems such as personal computers. Multiprocessors require careful data locality optimizations since accesses from other processors can lead to invalidations and false sharing cache misses. This thesis explores hardware and software approaches for creating and exploiting temporal and spatial locality in multiprocessors.

We propose the capacity prefetching technique, which efficiently reduces the number of cache misses but avoids false sharing by distinguishing between cache lines involved in communication from non-communicating cache lines at run-time. Prefetching techniques often lead to increased coherence and data traffic. The new bundling technique avoids one of these drawbacks and reduces the coherence traffic in multiprocessor prefetchers. This is especially important in snoop-based systems where the coherence bandwidth is a scarce resource.

Most of the studies have been performed on advanced scientific algorithms. This thesis demonstrates that a cc-NUMA multiprocessor, with hardware data migration and replication optimizations, efficiently exploits the temporal locality in such codes. We further present a method of parallelizing a multigrid Gauss-Seidel partial differential equation solver, which creates temporal locality at the expense of increased communication. Our conclusion is that on modern chip multiprocessors, it is more important to optimize algorithms for data locality than to avoid communication, since communication can take place using a shared cache.

Keywords: data locality, temporal locality, spatial locality, prefetching, cache, cache behavior, cache coherence, snooping protocols, partial differential equation, shared-memory multiprocessor, chip multiprocessor, simulation

Dan Wallin, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden

© Dan Wallin 2006

ISSN 1651-6214

ISBN 91-554-6555-2

urn:nbn:se:uu:diva-6837 (<http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-6837>)

Till pappa

List of Papers

This dissertation is based on the following papers, which are referred to by the capital letters A to F.

- A. Dan Wallin and Erik Hagersten, **Miss Penalty Reduction Using Bundled Capacity Prefetching in Multiprocessors**, In Proceedings of the International Parallel and Distributed Processing Symposium, Nice, France, April 2003.
- B. Dan Wallin and Erik Hagersten, **Bundling: Reducing the Overhead of Multiprocessor Prefetchers**, In Proceedings of the International Parallel and Distributed Processing Symposium, Santa Fe, USA, April 2004.
- C. Dan Wallin, Henrik Johansson and Sverker Holmgren, **Cache Memory Behavior of Advanced PDE Solvers**, Technical Report 2003-044, Department of Information Technology, Uppsala University, Uppsala, Sweden, 2003.
A shorter version of this paper was published in *Parallel Computing: Software Technology, Algorithms, Architectures and Applications*, volume 14, pages 475-482, 2004.
- D. Sverker Holmgren, Markus Nordén, Jarmo Rantakokko and Dan Wallin, **Performance of PDE Solvers on a self-Optimizing NUMA Architecture**, *Journal of Algorithms and Applications*, volume 17(4), pages 285-299, 2002.
- E. Dan Wallin, Henrik Löf, Erik Hagersten and Sverker Holmgren, **Multigrid and Gauss-Seidel Smoothers Revisited: Parallelization on Chip Multiprocessors**, Technical Report 2006-018, Department of Information Technology, Uppsala University, Uppsala, Sweden, 2006.
Accepted for publication in the Proceedings of the International Conference on Supercomputing, 2006.
- F. Dan Wallin, Håkan Zeffer, Martin Karlsson and Erik Hagersten, **Vasa: A Simulator Infrastructure with Adjustable Fidelity**, In Proceedings of the International Conference on Parallel and Distributed Computing and Systems, Phoenix, USA, November 2005.

Reprints were made with permission from the publishers. All papers are verbatim copies of the original publications but are reformatted to the one column format of this book.

Comments on my Participation

- A. Miss Penalty Reduction Using Bundled Capacity Prefetching in Multiprocessors**
I am the main contributor and the principal author of this paper. I implemented and performed all experiments.
- B. Bundling: Reducing the Overhead of Multiprocessor Prefetchers**
I am the main contributor and the principal author of this paper. I implemented and performed all experiments.
- C. Cache Memory Behavior of Advanced PDE Solvers**
Henrik Johansson and I are the main contributors of the paper. I am one of the co-authors of this paper. Henrik Johansson performed all experiments. I implemented the prefetch mechanisms in the simulator.
- D. Performance of PDE Solvers on a self-Optimizing NUMA Architecture**
All authors have contributed to the project. I am one of the co-authors of this paper and performed all the experiments on the pseudospectral solver kernel.
- E. Multigrid and Gauss-Seidel Smoothers Revisited: Parallelization on Chip Multiprocessors**
I am the main contributor and one of the co-authors of this paper. I parallelized the Gauss-Seidel kernel and performed all experiments. Henrik Löf is the main contributor of the multigrid code.
- F. Vasa: A Simulator Infrastructure with Adjustable Fidelity**
All authors have contributed to all aspects of the project. I am the principal author of the paper together with Martin Karlsson. I designed and performed all the experiments. Håkan Zeffner is the main contributor of the Vasa implementation.

Acknowledgements

It is already year 2006... When I begun as a Ph. D. student that seemed to be far in the future, but now I am here. Well, it feels nice to come to an end. However, the feelings are mixed. I have had a great time here at the IT department at Uppsala University. There are a lot of nice people who have helped me and made my time enjoyable.

First of all I would like to thank my two supervisors Erik Hagersten and Sverker Holmgren. It has been excellent working with you! You supervise in two very different ways and I believe that this combination has been excellent for me. We have had quite a few interesting meetings and discussions during the years, leading to new ideas. You have also spent many late evenings reading, editing and commenting my papers.

The members of the Uppsala Architecture Research Team (UART) have given me excellent support both at work-related, and even more importantly, social activities. I would like to express my gratitudes to Lars Albertsson, Erik Berg, Henrik Löf, Martin Karlsson, Zoran Radović and Håkan Zeffer. Working days would not have been at all as enjoyable without the thorough discussions on numerous subjects such as music, HiFi-equipment, sailing, cross-country skiing, ski wax, motorcycles etc. Captain Erik Hagersten has brought us all on sailing trips every year. Eight computer architects, with an average length of 1.90 m, made it somewhat crowded, but it was a lot of fun and I hope the tradition will continue!

Erik Berg, Joachim Parrow and I developed a new course in computer architecture a couple of years ago. We were a great team, helped each other and had a lot of fun. I think this was an excellent way of teaching this course.

I would like to thank all the co-authors I have had in my papers: Henrik Johansson, Martin Karlsson, Henrik Löf, Markus Nordén, Jarmo Rantokooko and Håkan Zeffer. Many other people contributed to my work: Karin Hagersten has been proofreading some of the papers, Björn Victor answered many Tex-questions and Jim Nilsson created the Sumo cache coherence model, which I used in my first papers.

Last year was a very difficult time for my family. Within a few months I suddenly lost my dad and my grandma. Things will never be the way they used to be. I am grateful to have a strong mom and sister, who both have managed to keep the mood. I would like to thank all friends and relatives for helping us and taking care of us in a hard time.

Most importantly, I would like to thank my fiancé and soon to be wife, Maria, for love, support and sympathy at all times. I love you!

Svensk sammanfattning

Utvecklingen inom datorområdet har som de flesta vet varit otroligt snabb. Detta har skett främst tack vare framsteg inom halvledarindustrin som har gjort det möjligt att få plats med allt fler transistorer på varje chip. Datorerna och elektronikkretsarna har blivit snabbare genom förminskningen av komponenterna men även i och med utvecklingen inom datorarkitektur.

Utvecklingen har dock inte gått lika fort för all datorkomponenter. Två viktiga delar av en dator är processorn, som bland annat utför alla beräkningar och jämförelser, och minnet där data lagras. I början av 80-talet tog det lika lång tid att hämta data från minnet som det tog att addera två tal i processorn. Idag kan en minnesåtkomst ta flera hundra gånger längre än en addition. Minnesteknologin har också förbättrats men inte i lika hög grad som processorteknologin.

Skillnaden mellan beräkningstid och minnesåtkomsttid medför stora problem för den som vill tillverka en snabb dator. En populär lösning är att använda cacheminnet. Ett cacheminne är ett mycket snabbt minne som byggs med en snabbare teknologi (SRAM) än det vanliga primärminnet (DRAM). SRAM kräver fler transistorer än DRAM-kretsar och är betydligt dyrare att tillverka. Cacheminnet placeras mellan processorn och primärminnet. När processorn behöver data för en beräkning kopieras datan från primärminnet till cacheminnet innan det skickas till processorn. Om processorn sedan behöver samma data igen, så finns det redan i cacheminnet, vilket minskar minnesåtkomsttiden. Tyvärr kan cacheminnet av ekonomiska skäl inte göras tillräckligt stort för att innehålla alla data, vilket gör att om cacheminnet är fullt kommer viss data att sparkas ut för att ge plats åt ny data.

För att få ut maximal prestanda bör ett program utformas så att det utnyttjar cacheminnet på bästa möjliga sätt. Detta görs bäst genom att utnyttja så kallad datalokalitet. Datalokalitet finns i två sorter, temporal lokalitet och spatial lokalitet. Temporal lokalitet innebär att man försöker utnyttja data så många gånger som möjligt innan man sparkar ut det ur cacheminnet. Med spatial lokalitet menas att adresser som ligger nära tidigare efterfrågade adresser med stor sannolikhet kommer att utnyttjas senare av processorn. En vanlig metod för att utnyttja detta är att man hämtar lite mer data till cacheminnet än det som har efterfrågats vid varje minnesåtkomst. Cacheminnet har visat sig vara en effektiv metod för att förbättra minnesåtkomsttiden. Därför har man idag ofta flera nivåer av cacheminnet i varje datorsystem. De snabbaste cacheminnet är minst och finns närmast processorn och de långsammaste cacheminnet är störst och placeras närmast primärminnet.

I stora datorsystem, som bland annat används för avancerade numeriska beräkningar eller för att hantera stora databaser, används datorer med många processorer. Dessa multiprocessorsystem byggs ofta med så kallat gemensamt minne. Gemensamt minne innebär att alla processorer har en och samma logiska bild av ett enda stort minne i systemet. Detta behöver dock inte betyda att allt minne fysiskt finns på bara en plats i systemet. Ofta delas minnet upp mellan de olika processorerna, vilket gör att en del av minnet ligger nära och en del långt bort sett ur varje processors perspektiv. Den del av minnet som finns närmast går ofta att komma åt betydligt snabbare än det minne som ligger längre bort.

Att designa system med gemensamt minne är svårare än att designa ett enkelprocessorsystem. Detta beror bland annat på att även multiprocessorsystemet använder cacheminnen för att förbättra minnesåtkomsttiden. Vissa av dessa cacheminnen är privata för varje processor. När ett multiprocessorsystem utnyttjar flera processorer för att köra ett program händer det att samma data kommer att finnas i flera kopior i olika cacheminnen i datorn. När en processor uppdaterar data gäller det att även de andra processorerna är medvetna om att en uppdatering skett i systemet för att undvika felaktiga beräkningar. Att hålla en samstämmig bild av minnet för alla processorer brukar kallas cachekoherens.

De senaste åren har det blivit populärt att bygga så kallade chip-multiprocessorer (CMPer). Den snabba utvecklingen inom halvledarkretsar och datorarkitektur har gjort det möjligt att placera flera processorkärnor på ett och samma datorchip. CMPer har fördelen att kommunikationen mellan de olika processorerna kan ske extremt snabbt. Anledningen till detta är att man inte behöver skicka några meddelande utanför chipet när man vill kommunicera utan att man helt enkelt kan kommunicera genom att skriva data till ett gemensamt cacheminne.

Denna avhandling beskriver olika sätt att skapa och bättre utnyttja datalokalitet. Avhandlingen beskriver både metoder som bygger på att designa hårdvaran på ett annorlunda sätt och metoder som går ut på att skriva effektiva program i mjukvara.

En metod för att minska minnesåtkomsttiden i ett datorsystem är att hämta data från minnet innan det ska användas. Denna metod brukar kallas för *prefetching*. För att lyckas hämta rätt data använder man någon typ av metod för att gissa vilket data som kommer att efterfrågas närmast. Tyvärr misslyckas ofta denna gissning vilket leder till att felaktig data hämtas till cacheminnet. Detta leder till en onödig datatrafik men också till att cacheminnet utnyttjas sämre eftersom värdefull data riskerar att kastas ut. De två första artiklarna, A och B, beskriver en metod, *bundling*, för att minimera datatrafiken genom att kombinera anrop efter flera data. Artikel A föreslår även en ny prefetching-metod som enbart hämtar den efterfrågade

adressen utan även ett antal nästkommande adresser under förutsättning att den ursprungliga adressen inte redan finns i cacheminnet. Detta gör att man undviker att spekulativt hämta data som används för kommunikation mellan flera processorer. Risken är nämligen stor att ej efterfrågad data kommer att studsas fram och tillbaka mellan processorerna och på det sättet förstöra för varandra.

Artikel C handlar även den till viss del om bundling-tekniken men för en helt annan typ av program nämligen tre avancerade ekvationslösare för partiella differentialekvationer (PDEer). I artikel C används simuleringsteknik för att undersöka prestandan hos de studerade lösarna, vilket väldigt få har gjort tidigare. Det simulerade hårdvarusystemet är helt uppbyggt i ett simuleringssystem. Det innehåller precis som det riktiga multiprocessorsystemet processorer, cacheminnet och datornätverk. Den simulerade miljön har dock bl a fördelen att det går att studera hur ett program kommer att uppföra sig innan motsvarande hårdvarusystem finns på marknaden. Det är också mycket enklare att få prestandainformation om cacheminnet, nätverket och processorn. Nackdelen är att simulering tar betydligt längre tid än verklig körning. Att simulera ett koherent multiprocessorsystem med flera nivåer av cacheminnet kan ta mer än 100000 gånger längre tid än på motsvarande hårdvara beroende på hur noggrant man simulerar. En annan nackdel är att det kan vara svårt att verifiera att simulatoren stämmer väl överens med verkligheten. Artikel F presenterar simulatoren Vasa som vi designat för att vara ett flexibelt sätt kunna simulera multiprocessorsystem med olika cachehierarkier och nätverk. Vasa är ett tillägg till en kommersiell mjukvara som kallas Simics som gör det möjligt att köra riktiga användarprogram på ett riktigt operativsystem. En stor fördel med Vasa är att den kan köras med flera olika noggrannhetsnivåer. Ju noggrannare man simulerar desto längre tid tar normalt simuleringen. Detta gör det möjligt för användaren att själv välja en lämplig noggrannhet beroende på problem. Det är inte heller säkert att man får en bättre överensstämmelse med verkligheten om man simulerar med hög noggrannhet eftersom man då ofta kör de simulerade programmen under kortare tid. Det går också att snabbspola förbi delar av koden i en mindre noggrann simuleringsnivå för att sedan återigen öka noggrannheten.

Optimeringarna som beskrivits i artikel A, B och C beskriver framförallt hur man kan bygga hårdvara för att utnyttja spatial lokalitet. Det går dock även att bygga ett system även för temporalt lokalitetsutnyttjande. Ett exempel på detta beskrivs i artikel D i Sun Wildfire-systemet. För att minska minnesåtkomsttiden i multiprocessorer allokeras ofta data i en del av primärminnet som är närmast den processor som först rörde vid data. Detta är oftast bra eftersom det är troligt att samma processor kommer att utnyttja samma data igen. Dock händer det att en annan processor börjar utnyttja samma data betydligt flitigare än den som allokerade data. I detta fall vore det fördelaktigt att flytta

datan till minnet närmast den nya processorn eftersom den ofta har snabbare minnesåtkomsttid. I Wildfire finns hårdvarustöd för att göra detta. Det minne som innehåller den efterfrågade datan kan antingen kopieras eller flyttas till den nya processorns minne. Artikel D undersöker hur väl detta fungerar för tre olika PDE-lösare. Resultaten visar att detta hårdvarustöd fungerar bra och att resultatet av en felaktig minnesplacering åtgärdas under körning.

Den sista artikeln, artikel E, handlar om att effektivt parallellisera en multi-grid PDE-lösare för Poissons ekvation på CMPer. I artikeln föreslås en parallellisering av problemet som i och för sig leder till mera kommunikation än tidigare använda metoder, men som har fördelen att den utnyttjar cacheminnet betydligt effektivare. Detta sker genom att man återutnyttjar data betydligt fler gånger innan man kastar ut det från cacheminnet, dvs den temporala lokaliteten är bättre. Nackdelen med metoden, dvs att mängden kommunikation ökar, är dock inget större problem på en CMP eftersom kommunikationen mellan två trådar på samma chip kan ske billigt via det gemensamma cacheminnet.

Contents

1	Introduction	17
1.1	Computer Development	17
1.2	Cache-Coherent Multiprocessors	19
1.3	Exploiting Locality	21
1.4	Parallel Programming	23
1.5	Writing Efficient Software	24
1.6	Application Behavior	25
1.7	Adaptivity	26
1.8	Evaluation of Data Locality	27
1.9	Contributions	28

Introduction

The computer development during the last decades has made memory references relatively more expensive than computations. To overcome the long memory latencies, cache memories have been introduced. A cache is a small, expensive, but fast, scratchpad-like memory located between the processor and the main memory. Since large caches often have longer access time than small caches, a hierarchy of caches with different ratio between speed and size is common in modern computers. The cache content is typically not directly managed by the program, why its existence is logically transparent to the programmer. However, the likelihood of finding the requested data in the cache often has a large impact on the execution speed of a program.

This thesis is about different ways of impacting the cache contents to make the program run faster. More specifically, systems consisting of several processors and caches, which share a common memory, are studied. In such systems, not only the cache content but sometimes also the data placement is important factors for performance. Studies in three different areas have been performed. First, the interplay between different existing run-time optimizations and applications is studied. Second, new run-time optimizations targeting cache prefetching are proposed and evaluated. Last, new ways of writing parallel algorithms are proposed and evaluated.

1.1 Computer Development

A famous prediction in computer development was made in 1965 by Gordon Moore at Intel [53]. He discovered that the number of transistors per area that was economically achievable had doubled every year to that date. He predicted that this development would continue for at least another ten years. The so-called “Moore’s law” has since then been reformulated and now is said to predict that the number of transistors will double every 18 month. This development has been driven by developments in process technology in the semi-conductor industry.

As a consequence of the shrinkage of the electronic components together with developments in computer architecture, the processors can operate at much higher frequencies than before. However, the memory access time has not improved as fast as the processor frequency. In the early 80’s, a memory reference took about the same time as an addition. Today, memory accesses can take several hundred times longer than a single addition. This makes memory references the bottleneck for many applications. To compensate for long

memory access latencies, cache memories have been introduced. To achieve maximum performance, it is important to use the cache memory efficiently and access cached data as much as possible.

There are a number of trends in processor development at the moment. The most apparent is that the clock frequency is not longer increasing as fast as it used to due to heat problems and power considerations. This is caused by the physical property which makes the dynamic power consumption increase with the square of the frequency. Also, the high frequencies has made the time it takes to send a signal on the chip a bottleneck, rather than the transistor switch time [7].

One popular way to gain further performance without increasing clock frequency is to increase the cache size and to add several cache levels. This is a rather simple measure, since the available transistors on a chip still increases rapidly. Larger caches give better performance as long as the algorithms can exploit locality well. However, larger caches usually have longer access latency and therefore modern computers often have a hierarchy of several cache levels. The largest cache in the hierarchy is normally slowest and located closest to the memory, while the smallest and fastest cache is located next to the processor. A larger fraction of the chip size has been dedicated for caches built from SRAM-technology during the last years of development.

Table 1.1 shows some technical data of server processors available to the market in March 2006. As can be seen, most processors today have roughly a quarter of a billion transistors. The next generation of Itanium 2 processor (*Montecito*), which is planned to be released in 2006, will have 1.72 billion transistors, 26 MB of on-chip caches and is predicted to run at 1.7 GHz [49]. This example clearly shows that the number of transistors and the cache size increase but the frequency is more or less unchanged.

It has also become popular to let each processor execute more than one thread. The two most commonly used implementations are chip multiprocessors, CMPs [13, 58], and simultaneous multithreading, SMT [25]. CMPs are built by putting several independent cores on a single processor die. The first generation of CMPs often integrates a number of single-core processors with private caches on the same chip, whereas the second generation of CMPs usually share caches between cores. SMTs let several threads share the resources and pipeline of a single core. This is possible since modern processors are pipelined and have several independent units that can perform arithmetical and logical tasks as well as memory operations concurrently. For maximum performance, as many of these units as possible should perform useful tasks each clock cycle. In many cases, dependencies between the tasks and long memory access times lead to pipeline stalls. By letting the processor execute several threads, there is a better chance of finding useful tasks for all units of the processor.

Processor	AMD Opteron	IBM Power 5+	Intel Itanium 2	Intel XeonMP	Sun UltraSPARC IV+	Sun UltraSPARC T1
ISA	x86-64	PPC	IA64	x86-64	SparcV9	SparcV9
Cores/die	2	2	1	1	2	8
Threads/core	1	2	1	2	1	4
Frequency (GHz)	2.8	1.9	1.66	3.66	1.5	1.2
L1 d cache (KB)	64	32	16	8	64	8
L1 i cache (KB)	64	64	16	12	64	16
L2 cache (MB)	1	1.92	0.25	1	2	3
L3 cache (MB)	-	36	9	1	32	-
Mem bw (GB/s)	6.4	12.8	10.6	5.3	4.8	25.6
Transistors (M)	233	276	592	125	295	279
Max power (W)	95	120	130	140	90	79

Table 1.1: Technical comparison of commercially available server processors in March 2006.

The CMP and SMT trends can also be observed among the processors in Table 1.1. The most extreme multithreaded processor is the Sun UltraSPARC T1 processor, which integrates eight processor cores each with four threads on a single chip. All major server processor vendors are releasing CMPs or SMTs at the moment [44, 46, 47, 48, 52]. Recently, also processors for smaller systems such as personal computers are adopting the CMP and SMT trend to gain further performance without increasing the clock-speed.

1.2 Cache-Coherent Multiprocessors

Multiprocessors have traditionally been used for demanding computations in commercial and scientific settings. Examples of such systems have been used for weather predictions, car crash simulations and protein modeling as well as running databases for online reservations and bank accounts. These systems are large expensive systems. The introduction of multithreaded proces-

sors also in cheaper systems makes it important to parallelize programs for maximum performance in many more applications than earlier.

From the programmer's point of view, shared-memory programming is the most intuitive programming model on a multiprocessor. In such a model, hardware gives the programmer the view of the entire multiprocessor system having a single shared memory. Keeping the memory coherent is a non-trivial task for the computer designer. The main reason for this difficulty is the use of caches to speed up data accesses. A schematic picture of such a system can be seen in Figure 1.1. In this kind of system, several copies of the same data

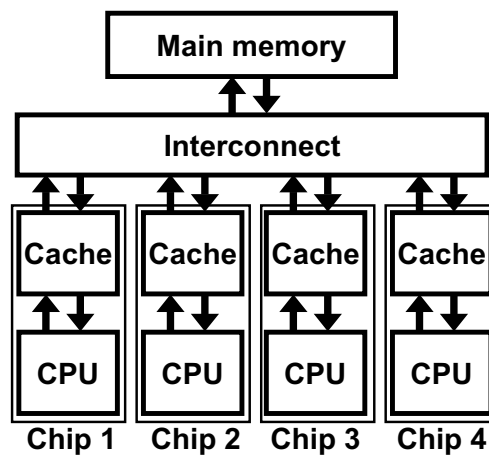


Figure 1.1: Symmetric multiprocessor

address can be found within the system. Since data are brought from main memory to the accessing processor, a certain piece of data can be found in several caches as well as in the main memory. It becomes complicated to keep the memory coherent as soon as an address is updated by one of the processors. In this case, the system has to make sure that the other processors get the same view of the memory also after the new value has been written.

Two major coherence protocol implementations are invalidate and update protocols [19]. In an update protocol, the system ensures coherence by updating all other processors' cached copies as soon as a processor writes a value. The invalidate protocol sends out invalidation requests to all other processor caches when a processor writes a value. The invalidate protocols have become most popular since they normally generate less traffic than update protocols.

A popular implementation of invalidation-based coherence is the Illinois protocol [10]. In the Illinois protocol, a cache line can be in any of four different states: modified, exclusive, shared and invalid, MESI. Modified indicates that the cache line has been written by a processor and that there is only one cached copy. Exclusive means that only this processor has a copy of the

cache line but it has not been modified. Shared has the meaning that two or more processors may have a copy in their respective caches. An invalid cache line does not contain valid data. An extension to this problem is the MOESI-protocol [63], which in addition to the previous states, also has an owner state. A cache line in the owner state contains shared data, but will take over the responsibility for supplying data at requests from the main memory.

Shared-memory systems implement coherence using either directory- or snooping-based protocols [19]. In directory protocols, parts of the memory are used to keep track of where the copies are stored, whereas snooping-based systems broadcast messages at all coherence changes. Directory-based coherence protocols are more scalable than snooping-based coherence, but snooping-based systems have a superior cache-to-cache transfer time. Another approach to coherence is to implement a software layer to handle the coherence, this is done in so-called software distributed shared-memory, SW-DSM [14, 43, 59, 61]. However, because of performance drawbacks these protocols have so far mostly been of academic interest and very few have been used in commercial products.

Shared-memory systems can have different topology in their designs. The topology depends on how the processors and the memory are distributed in the system. Two main categories of shared-memory multiprocessors are SMPs (symmetric multiprocessors) and cc-NUMAs (cache-coherent non uniform memory architectures). An alternative name of SMP is UMA (Uniform Memory Architectures). On a SMP-system the access time to all parts of the memory is the same for all processors, while the cc-NUMA have larger access times to remote than to local memory. It is therefore important to allocate memory correctly on a cc-NUMA for good performance.

Earlier SMPs were built using separate processors connected via off-chip high-speed interconnects. However, the computer development has made it efficient to put several processor cores on a single chip. In chip multiprocessors (CMPs), caches can be shared by several cores as shown in Figure 1.2. The shared caches can make the communication between threads executing on different cores on the same chip fast.

1.3 Exploiting Locality

The key to good performance on modern computers is fast access to data. As described above, caches in one of more levels are used to achieve this. It is important to use the available cache space efficiently since every cache miss can lead to long stall times for the processor. The trends in computer development make it even more important to avoid memory accesses, since they are becoming relatively more expensive than other instructions. To write

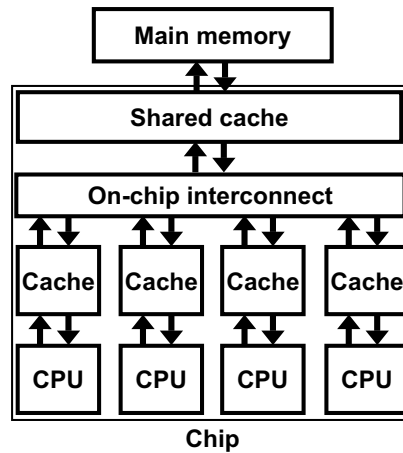


Figure 1.2: Chip multiprocessor

efficient programs, it is therefore important to understand why cache misses occur. Cache misses are often divided into four categories [23, 38]:

- **Compulsory misses** The first time the processor accesses a certain piece of data a mandatory compulsory miss occurs. The data is not yet available in a cache. These cache misses are often also called cold misses.
- **Capacity misses** The cache is not large enough to hold the entire data set of the application. The requested piece of data has been replaced to memory.
- **Conflict misses** All pieces of data cannot be stored in the cache contemporary. Conflict is caused by a non-perfect cache implementation.
- **Coherence misses** Coherence cache misses only occur in multiprocessors. They are caused by coherence activities such as a cache line being invalidated by another processor. Coherence misses can be divided into true and false sharing misses. True sharing misses occur whenever a piece of data is not available in a processor's own cache since another processor is doing useful work on the same piece of data. False sharing misses occur when an accessed address in a cache line is not available in a processor's local cache since another processor has updated another independent address in the same cache line.

There are several ways to avoid cache misses in computer designs. All of them have certain costs associated to them. Capacity misses are most easily avoided by increasing the cache size. This requires that a larger amount of the available chip area is reserved for cache space. Conflict misses can be avoided by building the cache with higher associativity, which unfortunately makes them more complex and can increase the cache access latency.

One of the standard approaches to improve cache miss rate is to build caches with larger cache line size. This makes it easier to exploit spatial locality

since a larger chunk of data surrounding the accessed address is brought to the cache. The probability is large that the cache contains the next accessed address because of the spatial locality property of programs. A drawback with large cache lines is that in a multiprocessor, large cache lines can lead to false sharing misses.

Prefetching is a technique to bring data from the memory to the cache in advance of the usage. Typical schemes are either hardware-based [12, 20, 21, 45, 64] or compiler-guided in software [54, 55, 56, 66]. Hardware prefetching schemes require a prediction mechanism that stores information on previous execution to correctly predict future data accesses. Incorrect guesses can lead to worse performance than in a non-prefetching system because of higher cache miss rates and bandwidth overhead causing contention.

Most compilers implement performance optimizations that can decrease the number of cache misses. Example of such optimizations are loop interchange, cache blocking, loop pipelining and unrolling [29]. Loop interchange makes loops execute in an efficient order so that spatial locality can be preserved. In cache blocking, the problem is reordered to fit the size of the caches. Loop pipelining and unrolling decreases the number of loop iterations and breaks dependencies between loops, which makes it easier for the processor to efficiently make use of all units in the hardware. These methods efficiently enhance performance for simple loops in structured codes but are often too difficult for a compiler to apply on complex scientific codes and commercial applications. In these cases, it is up to the programmer to make these code optimizations.

1.4 Parallel Programming

Parallelization is a non-trivial task and requires thorough understanding of the problem and careful programming. Nevertheless, the technology advances makes it necessary to write parallel codes for good performance. The most commonly used measure for evaluating the performance of a parallel code is speedup. The speedup S is defined as the ratio between the execution time on one thread, T_1 , and the parallel, T_p , execution time:

$$S = \frac{T_1}{T_p}. \quad (1.1)$$

Parallel programs do not normally show ideal linear speedup, that is, the speedup does not increase at the same rate as the number of processors. This is caused by parallel overheads, such as synchronizations, or code with certain serial components. These serial components lead to a scalability bottleneck of parallel programs expressed in Amdahl's law [8]. This law says that the

speedup of a program cannot be better than

$$S = \frac{1}{T_1^s + \frac{T_1^p}{p}}, \quad (1.2)$$

where T_1^s is the execution time of the serial part of the code on one thread and T_1^p is the execution time of the parallel part of the code on one thread. However, in some parallel programs super-linear speedup can be observed, that is, the speedup increases faster than the number of processors. This is usually an effect of hierarchical memory designs with caches. On a symmetric multiprocessor, Figure 1.1, the total amount of cache increases with the number of processors. The larger cache space can lead to better performance caused by faster memory references [34]. It should be noted that there is an alternative definition of speedup, where T_1 of Equation 1.1 is replaced with the best possible implementation of the code on one thread. It is therefore important to specify what kind of speedup that is computed.

On non-coherent distributed memory machines, a popular parallelization technique is to divide the problem into domains, which are located on independent computer nodes. The programmer in this case inserts commands in the code to explicitly send synchronization messages and data between the nodes. The Message Passing Interface, MPI [33], has become very popular for this kind of parallelization.

On cache-coherent shared-memory machines, the memory is shared between all nodes from the programmers' perspective. Data do not explicitly have to be passed between the processors, since the entire memory is shared. Multithreaded programs using for example Posix threads (Pthreads) [41] are common on these machines. Another popular way to parallelize programs on these machines is to insert OpenMP-directives in the code [18]. These directives are inserted by the programmer, for example surrounding a loop that is to be parallelized. An OpenMP-compiler then divides the work on several threads.

Many compilers support automatic parallelization. However, the difficulty for a compiler to find parallelism normally leads to a rather poor scalability for algorithms of normal complexity.

1.5 Writing Efficient Software

Writing an efficient algorithm requires thorough understanding of the problem to solve and is a multi-step procedure. First of all the solution method has to be chosen. The solution method can be based on own experience, on information found in literature or on a database-based expert system [40], which can give advice on how to solve a particular problem on a certain computer system. In

a multiprocessor environment it is important to consider whether the chosen method is possible to parallelize or not. Many problems are easier to solve using another method than in a sequential code, e.g., the red-black Gauss-Seidel iterative method studied in paper E.

It is also important to consider what program language to use. Some languages have built-in solutions for many problems that arise, resulting in a faster and simpler program. The development time of a program should also be considered together with the performance of the program. Some programming languages are better suited for good performance than others. It is also possible to take advantage of certain hand-tuned performance libraries available such as BLAS [16] and LAPACK [9].

After the language and solution algorithm has been chosen, a correct serial program should be written. It is at this time too early to worry too much about efficiency. Professor Donald Knuth at Stanford has in a famous quote stated: “premature optimization is the root to all evil in programming”. The results of the serial program should be verified to previous known results or using other means.

Now, when the serial program is correct. It is time to start optimizing. This can be done by using high optimization levels in the compiler or by explicit tuning by the programmer. Profiling tools, such as Gprof [31], Intel’s Vtune [4], Sun Microsystem’s Performance analyzer [6] and IBM’s Purify-Plus [3] could also be used to understand where in the application most time is spent and therefore is most important to optimize. Data locality is the key concept to performance on modern computers, so especially cache miss ratios are interesting to the programmer. The programmer could consider replacing certain pieces of the code with more cache-efficient codes or codes written in faster programming languages or performance libraries.

When the code has been optimized serially, it is time to parallelize the program using for example OpenMP or Pthreads, described in Section 1.4. The output of the parallel program should be compared to the results of the serial program to ensure correctness. Finally, the parallel program can also be optimized, taking into account for example load balancing techniques.

1.6 Application Behavior

Throughout the thesis, a large number of applications and benchmarks have been studied. The codes are used for solving many different problems in various areas. The programs are all shared-memory programs can be divided into three groups:

1. Simple technical, scientific and graphical applications from the SPLASH-2 suite [68]. The suite uses PARMACS-macros [11], which on our systems

are implemented on top of Pthreads. These are rather old, non-optimized codes to solve problems such as sorting, raytracing, numerical factorization etc.

2. Complex scientific codes used for solving partial differential equations (PDEs), which models many phenomena in nature. The solvers include kernels for the finite difference methods [17, 39], finite volume methods [24, 60] and spectral methods [27, 28] as well as a multigrid Gauss-Seidel solver [67]. All of these codes are parallelized using OpenMP.
3. Commercial workloads that models Java middleware and application servers, SPECjbb2000 [5] and ECperf [2] and a static web server, Apache [1]. These applications are parallelized with there own private threading libraries.

All these applications have very different characteristics in terms of cache misses, communication intensity, processor usage and memory footprints. Typically, commercial applications have a more random memory access pattern than scientific codes, which makes it difficult to exploit spatial and temporal locality efficiently [42]. The memory footprints can be large both for scientific and commercial applications. However, the largest market share is for commercial applications and therefore most computers are designed for maximum performance on these workloads. Many scientific codes put large pressure on other parts of the system, for example the available memory bandwidth and the processor pipeline, which can lead to degraded performance.

1.7 Adaptivity

The large differences in characteristics between the used applications and the underlying systems make it tempting to apply adaptivity. This can both be done in hardware by redesigning it to adapt to the changes of the applications at run-time and in software by rewriting the applications to suite different computer systems.

It is not a simple task to design a system in hardware that adapts to all acquirements of different applications. Building such a system requires careful tuning of the adaptive scheme since a small mistake can lead to worse performance than its static counterpart. Also, the hardware should not be too complex since its make it difficult to verify and often leads to a long time to market. Several authors have suggested adaptive schemes for multi-processor systems. Examples of such systems employ adaptive prefetching schemes [20, 30, 35, 64], access pattern detectors [32, 62] and adaptive coherence protocols [26, 37].

On NUMA-systems, the memory allocation can be very important for reasonable performance. The access time to remote memory is often two to ten times slower than the access time to local memory. In multiprocessors, locality can be improved by bringing the data to the cache of the processor frequently using that piece of data. This can be done by replicating the data between the processors or by migrating the data from a remote node to a local node. The Sun Wildfire system is one of the few systems that provide such hardware support [36]. Some operating systems provide system calls for migration of memory pages between nodes at run-time. It is up to the programmer to insert these calls in the programs to get reasonable performance in multithreaded programs [50].

The previously mentioned compiler optimizations are examples of adaptivity in software. The compiler can use information on important system parameters such as cache sizes, available processor units etc.

Adaptivity in algorithms can be exploited by ensuring load balance in parallel programs. This can be done in two ways, semi-statically or dynamically [19]. In a semi-static load balancing algorithm the scheduling is decided before run-time and a rescheduling is performed periodically. An example of a problem where this is useful is in scientific codes using adaptive mesh refinement [22]. In these problems the resolution varies in different areas of the solution, thus requiring a varying amount of computations. The dynamic load balancing uses a task-queue where new events are put. When a thread becomes available it can compute a task from the queue until all it has been emptied. This is very commonly used in commercial applications such as databases. The Standard Template Adaptive Parallel Library (STAPL) developed at Texas A&M University is also an example of adaptivity in software [65]. The library chooses among several solution methods at run-time to solve a problem optimally, depending on the problem size and the architecture.

1.8 Evaluation of Data Locality

Simulation can be used for evaluating and comparing computer architecture designs as well as for application studies. A major benefit of simulations is that it is possible to run applications on systems that are not yet available. This makes it possible to study new hardware designs and also to develop efficient software for future computers. The simulation software used in this thesis is the Simics full time simulator [51]. The user can extend the simulator with modules. These modules can for example model caches and interconnects.

Simics has the ability to model a uni- or multiprocessor system. It can boot a commercial operating system and run user programs. In this thesis we have simulated multiprocessor systems with the SPARC v9 ISA, which run the So-

laris operating system. The user applications have been installed on the modeled computer. Two different simulator frameworks have been used. These frameworks are both implemented using plug-in modules to Simics. In papers A, B and C, the SUMO simulator [57] has been used. Paper F introduces the Vasa simulator, which is more modular and implements more features than the SUMO simulator. The Vasa simulator is used in paper E.

Simulation requires a lot of computer resources, depending on the accuracy of the simulation. Often a single second of program execution on a real machine can require a hundred thousand or even a million seconds in the simulator. Also, a large number of design parameters and applications are normally studied in each experiment, where each combination of design and application requires a new run. Another drawback with simulation is that it is often complicated to verify that the model is correct. Therefore it is important to compare simulation results with for example analytical and statistical models. In paper E, the StatCache tool developed by Berg [15], which implements a statistical model is used in addition to simulations.

A fast way to get accurate information on for example cache references, cache misses and stalls is to use hardware counters. Hardware counters are available on most modern processors and gives the user valuable feedback at run-time. The results can be used to create a profile of the execution of a code using the tools mentioned in Section 1.5. A common approach is to give the user information on what parts of the code that take the longest to execute or cause the most cache misses. The drawback is that it is only possible to measure parameters on the hardware available and the information cannot be used to verify future hardware.

1.9 Contributions

The focus of this thesis is to propose ways of creating and exploiting data locality more efficiently. The papers in this thesis explore both hardware and software approaches to gain speed from spatial and temporal locality.

The hardware proposals, described in paper A and B, focus on exploiting spatial locality by using prefetching. The goal is to lower the cost of prefetching in multiprocessor systems. Paper C investigates the cache performance of three different PDE solvers using the ideas of paper A and B. Temporal locality can be preserved in a hardware system. A study of this has been performed on the Sun Wildfire system in paper D ¹. In paper E, a software optimization for temporal locality is proposed for a multigrid Gauss-Seidel PDE-solver. The thesis is concluded with a paper describing the Vasa simulator package,

¹Called Sun Orange in paper D

which provides an infrastructure for investigations of new hardware proposals and application optimizations.

In paper A, two new hardware prefetching optimizations are introduced: capacity prefetching and read bundling. It is often difficult to perform correct prefetches in multiprocessors, which can lead to additional coherence traffic on the interconnect. The proposed ideas aim to lower the cost of prefetching also in cases where the prefetching is not perfect.

Capacity prefetching classifies cache lines as communicating or non-communicating and only prefetches data for non-communicating cache lines. The simple detection method distinguishes these classes by looking at whether a requested cache line is available in the cache in the state invalid or if the cache line is not available. In the first case, the cache line is categorized as a communicating cache line, since another processor obviously has invalidated it. Cache lines that are not available in the cache are categorized as non-communicating. This very simple prefetch technique manages to reduce the number of cache misses and compared to many other prefetch techniques in multiprocessors it avoids false sharing misses.

The read bundling technique reduces address traffic and address snoops for read requests in hardware prefetchers in snooping multiprocessor systems. It can be used together with capacity prefetching for better overall performance. In paper B, the bundling technique is expanded to also include upgrade requests. The paper also investigates the performance together with other more complex prefetching schemes such as the adaptive prefetcher proposed by Dahlgren [20].

Bundling minimizes the number of bus requests by sending a single request for both the original as well as a number of prefetch addresses. The original request is extended with a bit-mask indicating the additional addresses to fetch. By bundling both the original and prefetch requests together into a single request for reads and upgrades, the address traffic can be kept under control. We also propose some changes to the coherence protocol implementation, which reduces the snoop lookups required by the caches. This is particularly important in snoop-based systems, since the number of snoop lookups that can be performed limits their scalability.

The bundling techniques work as follow:

- **Read bundling** On each read cache miss, a bundled read prefetch is generated for the original address as well as a number of prefetch addresses. When the owner of the original address is found the owner will respond with the original data as well as the prefetched data. However, the prefetched data will only be supplied if the responder is also the owner of the prefetch data.
- **Upgrade bundling** On each write miss generating an upgrade, bundled upgrade prefetches are generated for the original cache line and the prefetch

cache lines in the Shared or Owner state in the local cache. The upgrade request invalidates not only the original cache line but also the prefetch cache lines if there are exactly two copies in the system. An additional Owner state is needed in the coherence protocol to indicate if there is one or several simultaneously copies available.

Paper A and B studies the effect of bundling and prefetching on the SPLASH-2 benchmarks and two commercial workloads, SPECjbb2000 and ECperf. In paper C, similar studies have been carried out on advanced scientific codes.

Paper C, first of all shows that it is possible to perform simulation studies of realistically scaled numerical codes. The chosen codes are three kernels for solving PDEs. The kernels are taken from three different PDE-problems: computational fluid dynamics (CFD) [39], computational electromagnetics (CEM) [60] and computations of quantum dynamics (QD) [27]. Very few researchers have made studies on PDEs based on simulation techniques. As described in Section 1.8, simulation can be used to provide further knowledge to the programmer of the performance related to cache misses, bandwidth requirements, use of pipeline resources etc. This information is also possible to gain before the systems have been introduced on the market. The long simulation times and large computer resources required for simulation is often a smaller problem in numerical solvers since most of them are based on an iterative method. It is often enough to simulate one or a few steps in such a method and still get accurate results.

The results of paper C shows that large cache lines are beneficial in computers running these numerical solvers. Shorter cache line protocols using sequential prefetching show similar performance with regards to cache misses and data traffic.

In paper D, the behavior of the Sun Wildfire system has been analyzed running three advanced partial differential equation (PDE) kernels. The paper analyzes the effect of different memory allocation schemes, thread bindings and page migration and replication policies. The conclusion is that the self-optimizing features of the system manage to give performance similar to that of a system with an optimal initial data and thread placement. Paper D analyzes the performance of three PDE-solvers on the self-optimizing Sun Wildfire prototype system. The Wildfire system used in this study is a cc-NUMA that contains two nodes built from 16 processor SMPs. A software daemon detects pages, which are accessed in the remote node as candidates for migration to the local node. However, if threads in different nodes access the pages, space is allocated in both nodes. The system also provides support for memory placement. The custom policy is to allocate the data in the same node as the thread that first touched it was running on. The results of the study clearly show the benefit of introducing migration and replication features in

multiprocessors to exploit temporal locality.

We propose a method to make use of temporal locality in a multigrid Gauss-Seidel solver in paper E. The proposed algorithm reduces the amount of cache misses with an order of magnitudes compared to previous techniques at the expense of increased communication. We believe that the low cost of communication in chip multiprocessors make it necessary to reconsider the implementation of the algorithms. The effort of minimizing the amount of communication between threads is not most important for a chip multiprocessor environment. For these processors, it is better to focus on data locality also for multithreaded programs. The proposed temporally blocked Gauss-Seidel multigrid solver has a 60 percent speedup compared to a conventional multigrid solver based on the red-black ordering. Chip multiprocessors are different from previous SMP designs not only in the low cost of communication but also in the available cache space per thread. Since a large number of threads can share the same cache, the amount of cache per thread will probably decrease on future CMP designs. It is therefore even more important to use the cache resources carefully.

As previously mentioned, simulation can be used both for evaluating new hardware and software designs. The Vasa simulator infrastructure, presented in paper F, provides a modular way of performing such studies. Vasa provides most of the functionality of a modern computer system, including pipelined out-of-order SMT/CMP-processors, cache hierarchies and interconnects. It models timing and contention using two new concepts of timing, downstream timing and instant coherence, which makes the simulator easier to understand and a lot more efficient. Downstream timing models latencies and contention when requests are sent from the processor towards the memory system. Messages passed in the other direction take place momentarily. The instant coherence makes sure that all coherence changes within a multiprocessor simulation takes place instantly and all parts have the same global view of all states in the modeled system. The timing approximation of Vasa seems to strike a good balance between accuracy and performance for our needs. Vasa can be run in three modes with different levels of accuracy versus simulation speed. The user therefore can choose what accuracy that is useful for a certain experiment. It is also possible to fast-forward the simulation at uninteresting parts of the code by using a faster simulation mode. It should be noted that using the most detailed mode, does not always give the most accurate results. The reason for this is that a faster simulation mode might make it possible to cover larger parts of the code given a certain maximum simulation time. We have used Vasa in paper E to evaluate the performance of the temporal blocked Gauss-Seidel kernel. There is a remarkable correspondence between the results provided from hardware counters, statistical modeling using Stat-Cache [15] and the Vasa simulator package.

References

- [1] Apache, <http://www.apache.org/>.
- [2] Ecp perf, <http://ecperf.theserverside.com/ecperf/>.
- [3] IBM, PurifyPlus, <http://www-306.ibm.com/software/awdtools/purifyplus/>.
- [4] Intel, Vtune,
<http://www.intel.com/cd/software/products/asmo-na/eng/vtune/index.htm/>.
- [5] SPECjbb2000, <http://www.spec.org/osg/jbb2000/>.
- [6] Sun Microsystems, Performance Analyzer,
http://developers.sun.com/prodtech/cc/analyzer_index.html.
- [7] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the International Symposium on Computer Architecture*, 2000.
- [8] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computer capabilities. In *AFIPS Conference Proceedings*, 1967.
- [9] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, 1999.
- [10] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.
- [11] E. Artiaga, X. Martorell, Y. Becerra, and N. Navarro. Experiences on Implementing PARMACS Macros to Run the SPLASH-2 Suite on Multiprocessors. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, 1998.
- [12] J.-L. Baer and T.-F. Chen. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the Conference on Supercomputing*, 1991.
- [13] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzky, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the International Symposium on Computer Architecture*, 2000.

- [14] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 1990.
- [15] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2004.
- [16] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petinet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.
- [17] H. Brandén. *Convergence Acceleration for Flow Problems*. PhD thesis, Uppsala University, Sweden, 2001.
- [18] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishing, 2000.
- [19] D. E. Culler and J. P. Singh. *Parallel Computer Architecture - A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.
- [20] F. Dahlgren, M. Dubois, and P. Stenström. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, 1995.
- [21] F. Dahlgren and P. Stenström. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):385–398, 1996.
- [22] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White. *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers, Inc., 2003.
- [23] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, 1993.
- [24] F. Edelvik. *Hybrid Solvers for the Maxwell Equations in Time-Domain*. PhD thesis, Uppsala University, Sweden, 2002.
- [25] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5):12–19, 1997.

- [26] B. Falsafi and D. A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the International Symposium on Computer Architecture*, 1997.
- [27] M. D. Feit, A. Fleck, and A. Steiger. Solution of the Schrödinger Equation by a Spectral Method. *Journal of Computational Physics*, 47(3):412–433, 1982.
- [28] F. Fornberg. *A Practical Guide to Pseudospectral Methods*. Cambridge University Press, 1998.
- [29] R. P. Garg and I. Sharapov. *Techniques for Optimizing Applications*. Sun Microsystems Press, A Prentice Hall Title, 2002.
- [30] E. H. Gornish. *Adaptive and Integrated Data Cache Prefetching for Shared-Memory Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.
- [31] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the Symposium on Compiler Construction*, 1982.
- [32] H. Grahn and P. Stenström. Evaluation of a Competitive-Update Cache Coherence Protocol with Migratory Sharing Detection. *Journal of Parallel and Distributed Computing*, 39:168–180, 1996.
- [33] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1999.
- [34] J. L. Gustafson. Fixed Time, Tiered Memory, and Superlinear Speedup. In *Proceedings of the Distributed Memory Computing Conference*, 1990.
- [35] E. Hagersten. *Toward Scalable Cache-Only Memory Architectures*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 1992.
- [36] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the Conference on Supercomputing*, 1999.
- [37] E. Hagersten, A. Landin, and S. Haridi. DDM - A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, 1992.
- [38] M.D. Hill and A.J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.
- [39] C. Hirsch. *Numerical Computations of Internal and External Flows*. Wiley, 1988.

- [40] E. N. Houstis, A. C. Catlin, J. R. Rice, V. S. Verykios, N. Ramakrishnan, and C. E. Houstis. PYTHIA-II: A Knowledge/Database System for Managing Performance Data and Recommending Scientific Software. *ACM Transactions on Mathematical Software*, 26(2):227–253, 2000.
- [41] IEEE Std 1003.1-1996, ISO/IEC 9945-1. *Portable Operating System Interface (POSIX)–Part1: System Application Programming Interface (API) [C Language]*, 1996.
- [42] M. Karlsson, K. Moore, E. Hagersten, and D. A. Wood. Memory System Behavior of Java-Based Middleware. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2003.
- [43] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conference*, 1994.
- [44] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, 2005.
- [45] D. M. Koppelman. Neighborhood Prefetching on Multiprocessors Using Instruction History. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2000.
- [46] K. Krewell. Power5 Tops on Bandwidth. *Microprocessor Report*, December 22, 2003.
- [47] K. Krewell. Double Your Operons; Double Your Fun. *Microprocessor Report*, October 11, 2004.
- [48] K. Krewell. SPARC Turns 90 nm. *Microprocessor Report*, October 25, 2004.
- [49] K. Krewell. Best Servers of 2004. *Microprocessor Report*, January 18, 2005.
- [50] H. Löf and S. Holmgren. Affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *Proceedings of the International Conference on Supercomputing*, 2005.
- [51] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hällberg, J. Högborg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.
- [52] J. McGregor. Ringside for 2006 Dual-Core Fights. *Microprocessor Report*, December 19, 2005.
- [53] G. E. Moore. Cramming more Components onto Integrated Circuits. *Electronics Magazine*, April 1965.

- [54] T. C. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *ACM Transactions on Computer Systems*, 16(1):55–92, 1998.
- [55] T. C. Mowry and A. Gupta. Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, 1991.
- [56] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.
- [57] J. Nilsson, P. Nandula, and A. Landin. *SUMO - A Generalized Memory Hierarchy Simulator*. Sun Microsystems Inc.
- [58] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.
- [59] Z. Radović and E. Hagersten. Removing the Overhead from Software-Based Shared Memory. In *Proceedings of the Conference on Supercomputing*, 2001.
- [60] Matthew N. O. Sadiku. *Numerical Techniques in Electromagnetics*. CRC Press, 2000.
- [61] D. J. Scales and K. Gharachorloo. Design and Performance of the Shasta Distributed Shared Memory Protocol. In *Proceedings of the International Conference on Supercomputing*, 1997.
- [62] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the International Symposium on Computer Architecture*, 1993.
- [63] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *Proceedings of the International Symposium on Computer Architecture*, 1986.
- [64] M. K. Tcheun, H. Yoon, and S. R. Maeng. An Adaptive Sequential Prefetching Scheme in Shared-Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, 1997.
- [65] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2005.

- [66] D. M. Tullsen and S. J. Eggers. Effective Cache Prefetching on Bus-Based Multiprocessors. *ACM Transactions on Computer Systems*, 13(1):57–88, 1995.
- [67] P. Wesseling. *An Introduction to Multigrid Methods*. John Wiley and Sons Ltd., 1992.
- [68] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

Acta Universitatis Upsaliensis

*Digital Comprehensive Summaries of Uppsala Dissertations
from the Faculty of Science and Technology 176*

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-6837



ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2006