Multicore computing—the state of the art

Karl-Filip Faxén[1](editor), Christer Bengtsson[2], Mats Brorsson[3], Håkan Grahn[4], Erik Hagersten[5], Bengt Jonsson[6], Christoph Kessler[7], Björn Lisper[8], Per Stenström[9], Bertil Svensson[10]

December 3, 2008

Abstract

This document presents the current state of the art in multicore computing, in hardware and software, as well as ongoing activities, especially in Sweden. To a large extent, it draws on the presentations given at the Multicore Days 2008 organized by SICS, Swedish Multicore Initiative and Ericsson Software Research but the published literature and the experience of the authors has been equally important sources.

It is clear that multicore processors will be with us for the foreseeable future; there seems to be no alternative way to provide substantial increases of microprocessor performance in the coming years. While processors with a few (2–8) cores are common today, this number is projected to grow as we enter the era of manycore computing. The road ahead for multicore and manycore hardware seems relatively clear, although some issues like the organization of the on-chip memory hierarchy remain to be settled. Multicore software is however much less mature, with fundamental questions of programming models, languages, tools and methodologies still outstanding.

1 Introduction

The background of the current trend towards multicore computing is well known. For many years, increases in clock frequency drove increases in microprocessor performance. The increasing gap between processor speed and memory speed

[1]Swedish Institute of Computer Science, kff@sics.se
[2]Swedsoft
[3]Royal Institute of Technology
[4]Blekinge Institute of Technology
[5]Uppsala University
[6]Uppsala University
[7]Linköping University
[8]Mälardalen University
[9]Chalmers University of Technology
[10]Halmstad University

was bridged by caches and instruction level parallelism (ILP) techniques. Exploiting ILP means executing instructions that occur close to each other in the stream of instructions through the processor in parallel. This can mean partial overlap as in pipelining (fetching one instruction while decoding the previous one and executing the one before that) or real side by side superscalar parallelism. Modern ILP processors can have several tens of instructions in various stages of processing at the same time.

Thus, for cache hits, latencies scaled with reductions in cycle time (a cache hit was always the same number of cycles, typically two or three, even when processor frequency increased) while misses were overlapped with other misses as well as useful computation using ILP.

This situation did however end a few years back, and in the last three or so years, clock frequencies have not increased at all. The main reason for that is power, which increases more than linearly with clock frequency, and causes problems with dissipating the heat generated in the circuit. There are also fundamental physical limits in MOS transistor geometry that puts a lower limit to switching speed while still allowing for physical shrinkage of the transistors.

For the reasons discussed above, performance improvements must come from sources other than increased clock frequency. Mainly, this must be increased transistor counts, as this is the major resource that is still projected to grow in the future. Increased numbers of transistors have been used in different ways.

- First, increased second level cache sizes improve hit ratios and reduce the performance losses due to cache misses. This approach is however limited by the amount of performance lost in misses in the second level cache (increasing the size of the first level cache is not a good trade-off as it would force a reduction in clock frequency). If half of the execution cycles are spent waiting for misses in the second level cache, increasing its capacity can never make the program more than twice as fast.

- Second, ILP can be more aggressively exploited by wider issue capacity and additional functional units. The downside is increased design complexity and diminishing returns since there is only a limited amount of exploitable ILP in single threaded code. That ILP has to be shared among multiple issue and multi cycle instruction latencies. If the average instruction latency is two cycles and two instructions are issued each cycle, only programs with an ILP of four or more will execute efficiently [12].

- A third possibility is to design an aggressive processor core and feed it instructions from several threads while also exploiting ILP within each thread. This technique is known as *simultaneous multithreading* (SMT) [23] and has been used in processors from IBM (Power 5 and 6) as well as Intel (where it is called Hyper Threading). This means that utilization can be increased even for cores with more hardware resources (functional units, ...) than can be efficiently exploited by a single thread and it has the advantage that a thread that has an unusual amount of ILP can exploit it using the abundant hardware resources of the core.

- A fourth possibility is to put multiple cores on a single die [17]. This organization has a similar programming model as SMT in that it is based on thread parallelism rather than (only) ILP. The advantage of the multicore organization is that multiple copies of a simpler core have big design complexity advantages over a single complex SMT core. Also, communication within an SMT core with a thousand functional units will necessarily have long latency and the wiring will take a very large part of the chip area. Many chips implement multiple identical SMT cores (the IBM Power 5 and 6) or have otherwise multithreaded cores (the Sun Niagara / Niagara 2).

2 State of the art and current challenges

2.1 Hardware

Currently, multicore processors are the norm for servers as well as desktops and laptops, while penetration in the embedded sector is more uneven. There are two broad classes of processors. First, there are those that contain a few very powerful cores, essentially the same core one would put in a single core processor. Examples include AMD Athlons, Intel Core 2, IBM Power 6 [14] and so on. Second, there are those systems that trade single core performance for number of cores, limiting core area and power. Examples include the Tilera 64, the Intel Larrabee [19]and the Sun UltraSPARC T1 [13]and T2 [11] (also known as Niagara 1-2).

For the future, there are a number of open issues.

2.1.1 Core count and complexity

This issue, where current commercial offerings are divided, hinges on the expected speedup from additional cores. For markets where a substantial fraction of the software is not parallelized, such as desktop systems, speedup from extra cores is less than linear and may frequently be zero. Hence a few copies of the most powerful core that can reasonably be designed is the preferred alternative and is realized by most chips in this domain.

If on the other hand the expected speedup from extra cores is assumed to be linear the situation is different. In this regime, core design should follow the KILL rule, formulated by Anant Agarwal: Kill If Less than Linear. This means that any (micro) architectural feature for performance improvement (out of order instruction issue, larger caches, hardware branch prediction, ...) should be included if and only if it gives a relative speedup that is at least as big as the relative increase in size (or power or whatever is the limiting factor) of the core. That is, if a feature gives 10% speedup for a 5% area increase it should be included, but if the price tag is a 15% area increase it should be left out. The KILL rule has guided Tilera to a core design for the Tilera 64 that is a three issue VLIW (Very Long Instruction Word) architecture.

State of the art There are currently two broad groups of designs. First, there are the more conservative designs that start from a core that maximizes single thread performance and then puts as many of those cores as fits on a single die (or at least in a single package). Typical examples are the Core 2 Duo and Quad, the AMD Phenom x4, IBM Power 4-6, Sun UltraSPARC IV. Second, there are more radical designs that trade less single thread performance against better aggregate performance. Best known in this category are probably the Sun UltraSPARC T1 and T2 (also known as Niagara) with the Tile64 as another example.

Current challenges Which way this trade off is going to evolve is mainly a function of the evolution of workloads. If highly parallel programs become the norm, more and simpler cores are to be expected, and the other way around.

2.1.2 Heterogeneity

In a multicore chip, the cores could be identical or there could be more than one kind of core. There are two levels of heterogeneity depending on whether the cores have the same instruction set or not. Hence there are three possibilities:

1. Identical cores, as in most current multicore chips from the Intel Core 2 to the Tilera 64.

2. Cores implementing the same instruction set but with different nonfunctional characteristics.

3. Cores with different instruction sets like in the Cell processor where one core implements the PowerPC architecture and 6-8 *synergistic processing elements* implement a different RISC instruction set.

The Sun Niagara 1 chip shares one floating point unit among eight cores in a software transparent way, a design that could be counted in the second category.

A homogeneous system has the advantage of being simpler to analyze and resource allocate than a heterogeneous system, and is also simpler to design since it is build out of just one kind of component which is duplicated across the chip. On the other hand specialized hardware is more area and energy efficient. Hence a truly heterogeneous system (the third option) offers a promise of increased performance if the software challenges can be mastered, especially in situations where the workload of the system is known in advance.

One potentially useful kind of heterogeneity is to have a small number of very fast (wide issue, out of order) cores for parts of the computation with limited parallelism and a large number of simpler cores designed according to the KILL rule to exploit abundant parallelism when it is available. In addition, as core counts grow and feature sizes shrink, process variations may also add heterogeneity in the clock frequencies supported by different cores.

State of the art Most designs targeting desktops, laptops and servers are homogeneous, but in the embedded sphere, heterogeneity is more common, evidenced by for instance the Cell processor and the typical architecture of mobile phones.

Current challenges For heterogeneous systems, programming tools remain a challenge as compared to on a homogeneous system.

2.1.3 Memory hierarchy

On chip bandwidth and processing power are large compared to off chip bandwidth while on chip latencies are correspondingly small. Thus some form of local memory is a ubiquitous feature of all multicore designs; otherwise it becomes impossible to feed more than a small fraction of the functional units that fit on a chip with operands. The organization of this local memory is not clear, though.

One possibility is to have explicitly addressed local memories, accessed either with special instructions or appearing as a special part of the address space. This design is realized in the Cell processor, where the synergistic processing elements each has a 256 KB local memory which typically communicates with system memory using DMA transfers.

Most multicore designs however provide some form of coherent caches that are transparent to software. First level caches are typically private to each core and split into instruction and data caches, as in the preceding generation of single core processors. Then the options fan out.

- Early dual core processors had private per core second level caches and were essentially a double single core processor with a minimum of glue logic and an essentially snooping coherence protocol. Some designs continue with separate L2 caches, like the Tilera 64 where each core has a 64 KB L2 cache. However, the glue logic is in this case anything but simple and amounts to a directory based cache coherency protocol on a mesh interconnect.

- Second level caches can be shared between the cores on a chip; this is the choice in the Sun Niagara (a 3MB L2 cache) as well as the Intel Core 2 Duo (typically 2-6 MB).

- Separate L2 caches backed by a shared L3 cache as in the AMD Phenom processor (512 KB L2 per core, shared 2MB L3) or the recent Intel Core i7 (256 KB L2 per core, shared 8MB L3).

- A hierarchy where L2 caches are shared by subsets of cores. This pertains to the four core Intel Core 2 Quad, which is essentially two Core 2 Duo in a single package. Each of the chips have an L2 cache shared between its two cores but the chips have separate caches.

The various cache designs have been historically dictated, like early dual core chips which were two single core chips side-by-side leading to private L2 caches, but also based on performance and manufacturing considerations (it is easier to get a high yield on two smaller dies, as in the Core 2 Quad, than one large). The performance trade off is the following:

With private L2 caches, the L1-L2 communication is local, and the inter-core interconnect is located below the L2 cache, whereas with a shared L2 it sits between the L1 and L2 caches. In the shared case, all L1 misses go over the interconnect whereas in the private case only those that also miss in the L2 do so. This requires a more expensive, low latency interconnect (often a crossbar) which uses a lot of area that could otherwise be used for larger caches (or more cores). Also, L2 access time is increased by the need to go over the interconnect.

On the other hand, private L2 caches might waste chip area by having the same data occupy space in several caches, and accessing data in the L2 of another core, something that is sometimes needed to achieve cache coherency, becomes more expensive than accessing a shared L2 cache. Interestingly, there are academic studies showing either approach to be superior. What seems clear, however, it that it will be increasingly difficult to maintain a shared L2 cache as the number of cores climb into the tens and hundreds, so the issue is not so much it we will see L2 caches shared between all cores on a chip as it is whether we will see them shared between subsets or whether the L2 cache will, just like the L1, become part of the core.

State of the art Per core first level caches are the norm, to the point where these are simply regarded as part of the core. Second level caches come in both shared (Intel Core 2 Duo, IBM Power 4–6) and private (AMD Phenom x4) varieties, as well as semi shared (Intel Core 2 Quad is really two dies in a package, with the cores on each die sharing a second level cache). Some systems, like the Phenom, complement private second level caches with a shared third level cache.

Current challenges The main challenge ahead is scaling to manycore systems with thousands of cores. Thus there is a need to both minimize the number of misses out of a core, since these will require traversing expensive interconnects that will necessarily have delays in the tens of cycles in the worst case, and to avoid misses out of the chip, which may, depending on packaging technology, use scarce memory bandwidth.

2.1.4 Interconnect

The cores on a die must be connected to each other, and there are several possibilities.

- Classical buses do not scale beyond a limited number of cores and are not used in current designs beyond a few cores.
- Rings are used both in the Cell processor and in the Intel Larrabee and fit well with snooping cache coherency protocols where memory transactions

need to be visible to every core unless they are entirely private to a single core. Essentially, rings have emerged as better buses due to the lower power and higher frequencies allowed by the shorter lines and simpler arbitration logic inherent in the ring architecture.

- Crossbars are used in for instance the Sun Niagara processors and offer low latency high band with interconnects that fit well with directory based cache coherency protocols.
- Switched networks, typically 2-D meshes like in the Tilera 64 or possibly (fat) trees.
- Hierarchical interconnects where groups of cores are interconnected in some way and groups of groups are interconnected in a possibly different way. For instance, cores could be interconnected in small groups using buses or rings and those groups could communicate with each other over a mesh.

It is quite clear that manycore processors will have neither buses, rings or crossbars. For buses, long lines give high power consumption and low speed. Crossbars scale as the square of the number of ports and thus become untenable. Rings scale in terms of area and power, but since each transaction must pass all cores, latency is linear in the number of cores and the interval in which cores are allowed to inject transactions also increase linearly.

This arguments leaves switched networks and hierarchical interconnects as the main contenders for the future. Note also that if cache coherency is to be supported, the coherency mechanism interacts heavily with the interconnect structure. A mesh network, for instance, fits naturally with a directory based coherency mechanism, whereas a hierarchical system could have snooping in the leaves using rings or buses and use directories between the groups.

State of the art Crossbars are often used in designs with few processors, but rings and meshes are becoming more common.

Current challenges Rings and buses fit well with snooping cache coherence protocols, but for meshes directory based protocols are needed, and they have some scaling issues. Here hierarchical organizations might help.

2.1.5 Memory interface

Traditionally, off chip interfaces have been placed along the periphery of the chip. This has the drawback that if the number of devices along the periphery increases linearly (using larger chips or smaller features), the total number of devices increases quadratically. Thus the memory bandwidth per core will decrease as we move to chips with more cores. It then becomes attractive to stack memory chips on top of processor chips and spread the connections over the area of the chips, providing better scaling and also substantially improving memory latency [15].

In this kind of technology, multiple dies are placed on top of each other and connected using *Through Silicon Vias* (TSVs). These give fairly dense signal connections between chips; pitches (distances between the connections) of only 4–10 um have been reported [15] which allows for a 1024 bit bus in only 0.32mm^2 (which is a really small fraction of the not unusual 200mm^2 area of a current chip). The chips are thinned to a few um in thickness; this has the interesting consequence that vertical distances become smaller than horizontal ones so that vertically aligned parts of different chips are closer together than horizontally distant parts of the same chip. In the long run, this might violate the assumption that in a multicore processor, all the other cores are closer than main memory.

State of the art High end processors have up to four DDR2 interfaces for an aggregate memory bandwidth of over 20GByte per second.

Current challenges The main challenge is to scale memory bandwidth with the number of cores. The scaling requirement might be made somewhat less strict if the increasing aggregate cache sizes that come with the scaling of the number of cores can be used to decrease off-chip cache miss ratios; see Section 2.2.5.

2.1.6 Number of threads per core

Cores could support different numbers of threads. Many multicore designs have single threaded cores, like the Intel Core 2, the AMD Athlon based processors and the Tilera 64 chip. Others have multithreaded cores like the Sun Niagara (four threads per core in Niagara 1 and eight in Niagara 2), the IBM Power 5 and 6 (two threads per core) and the Cell processor (2 threads in the PowerPC core).

Using multithreaded cores is a way of increasing the utilization of core resources that are often idle when cache misses or (in complex core designs) branch mispredictions occur. As such the performance improvements are bounded by the amount of underutilization of the single threaded core. Typically, a doubling of the number of threads in a core does not lead to a doubling of core performance since the threads compete for core resources such as functional units and, perhaps most importantly, cache space. Thus a multithreaded core might have more cache misses than a single threaded one but still deliver better performance by tolerating the misses better.

2.1.7 Instruction set extensions

Many techniques for meeting the challenge of ubiquitous parallelism involves instruction set extensions, and there is also more traditional multiprocessor support that is already implemented. In particular, instruction sets have for a long time included atomic read-modify-write instructions such as test-and-set which read a memory location and store a new value in it atomically, that is, in such a way that accesses from other processors either happen before the read

or after the write. Such instructions are at the heart of the implementation of traditional synchronization primitives like locks and semaphores.

More recently, nonblocking primitives such as compare-and-swap have been added. These primitives are similar to the above atomic instructions, but perform the store only if a condition is true. For compare-and-swap, the condition is that the value in the memory location is equal to a given value. This primitive can be used for implementing for instance an increment of a shared counter without locking by first reading the old value of the counter, then computing the increment and finally updating the counter with the new value only if it has not been changed in the meantime.

Transactional memory (see Section 2.3.1) is a generalization of nonblocking synchronization that can benefit from hardware, and thus instruction set, support. The same is true of thread level speculation, discussed in Section 2.3.2.

Cache coherent hared memory is a powerful way for processors to communicate, but it is also quite expensive since one core must write to its cache, for which it needs to be the only core caching that memory location, then notify the other core of the availability of data, then the other core must read the location and move it into its own cache. For this reason, some processors, such as the Tile-64, provide message passing instructions between registers in different cores, bypassing the memory for lower latency [25].

State of the art Current processors typically support both atomic read-modify-write instructions and nonblocking primitives of the read-modify-maybe-write variety. These can be used to implement nonblocking operations including software transactional memory. In addition, the Rock processor from Sun, due to be released in 2009, implements transactional memory [16].

Current challenges At this time, it is not clear what mark the multicore issue will leave in instruction sets, especially whether extensions such as transactional memory or message passing will become common. However, it appears that message passing can give a significant latency reduction in inter core communication.

2.2 Software

The multicore revolution is a software revolution. Not only does software need to adapt to the new environment by being parallelized, but parallelization makes the software more complicated, error prone and thus expensive. There is also no consensus as to which programming model to use with a spectrum of proposals from keeping the sequential model and using automatic parallelization to programming with a low level threads interface. In the latter case, debugging becomes much more difficult due to the inherently nondeterministic nature of multithreaded programming.

2.2.1 Programming models

There are several programming models that have been proposed for multicore processors. These models are not new, but go back to models proposed for multi chip multiprocessors.

- Shared memory models assume that all parallel activities can access all of memory. Communication between parallel activities is through shared mutable state that must be carefully managed to ensure correctness. Various synchronization primitives such as locks or transactional memory is used to enforce this management.
- Message passing models eschew shared mutable state as a communications medium in favor of explicit message passing. Typically used to program clusters, where the distributed memory of the hardware maps well to the models lack of shared mutable state.
- In between these two extremes there are partitioned global address space models where the address space is partitioned into disjoint subsets such that computations can only access data in the subspace in which they run (as in message passing) but they can hold pointers into other subsets (as in shared memory).

Most models that have been proposed for multicores fall in the shared memory class.

Programming models also differ in whether they are directed towards computation[1] (parallelism only serves to enhance performance) or concurrency (parallelism is an essential part of the problem). The concurrency class can be considered more general in that it is typically possible to write a computational program in a concurrent language but not necessarily the other way around.

Kernel threads The lowest level shared memory programming model is *kernel threads*, long lived concurrent activities sharing mutable state, closely corresponding to the long lived cores sharing memory. The long lived nature of kernel threads comes from their implementation in the operating system, making thread operations such as creation and destruction relatively expensive, forcing them to be amortized over relatively long lifetimes. Because of this, it is often impossible to find large enough parts of a computation that is entirely independent of each other to allow the threads to be independent. Thus threads typically needs to synchronize with each other or ensure mutual exclusion using locks, condition variables or transactional memory.

Since kernel threads map so closely to the underlying hardware, they achieve (in the hands of an expert!) the best performance of the different models, and they are often used to implement the higher level models. In this way, kernel threads is the "assembly language" of multicore programming. Typical

[1] By computational we mean not only numeric computation but every case were some output is produced based on some inputs.

exponents of this model are `pthreads` that is commonly available in the Unix derivative operating systems, and Windows threads under Windows.

The thread model is in essence concurrent and can be used for concurrent as well as computational programming.

User level threads These are similar to kernel threads, but implemented in libraries and language run time systems, making them much less expensive. They can, depending on the implementation, have more or less exactly the same semantics as kernel threads. In particular, they can be pre-emptive, so that an infinite loop in one thread does not necessarily lead to nontermination of an entire program.

Examples include threads in concurrent programming languages such as Mozart and Erlang (where they are called processes).

The Erlang model differs from other models we have discussed by being based on message passing rather than shared mutable state. Erlang was created as a language for programming concurrent applications, notably telecoms equipment, but has since been used for distributed processing. While it is not specifically targeted towards multicore systems, recent implementation work has adapted the run time system to this environment, and allowed legacy code to seamlessly move to multicore platforms.

SPMD Single program multiple data (SPMD) is a programming model with its origins in high performance computing. It sits between thread level programming and task level programming in that there is a concept of threads, which execute the same code potentially with different data (hence the name of the model). These threads are implicit and similar to workers (see Section 2.2.4) although they are visible in that per thread data is available. The most well known example of this model is OpenMP.

Tasks Tasks differ from threads in that they are very light weight, always implemented in user mode. In some implementations, task creation can be accomplished in a few tens of cycles. Since they are cheap, they can be short lived enough that they can often run completely independently of each other, as for instance the iterations of a parallel loop.

Tasks are parallel, but in essence not concurrent. Thus they are not pre-emptive and can typically be executed sequentially. That is, creating a new task to perform a computation is semantically equivalent to performing the computation in a procedure call.

This property is exploited by high performance implementations of the task model such as Cilk, where most tasks are simply executed as procedure calls and only as many as are necessary to keep the hardware busy are actually executed in parallel. With version 3.0, OpenMP has also taken steps in the direction of task parallelism in this sense, although its heritage is more thread (SPMD style) oriented.

Data parallelism One of the major sources of parallelism in computational programs is operations over the elements of collections of data.

RapidMind and Intel's Ct are current examples of the data parallel paradigm, but the ideas go back at least to Fortran 90 and High Performance Fortran.

Sequential programming If an automatic parallelizing compiler is available, a multicore processor can be programmed just as if it were a sequential processor. This is often cited as the Holy Grail of compiler development, but like its namesake, this grail is elusive. For most sequential programs, only a little parallelism is found, and it is also difficult to utilize the parallelism efficiently. Looking at SPEC CPU numbers (were automatic parallelization is allowed) one sees that it has very little effect on average although it is effective for a few of the programs.

Some commercial compilers like Intel's `icc` provide automatic parallelization.

Domain specific programming languages Domain specific languages are tailored to a specific problem domain and embody knowledge of that domain. For instance, a constraint programming language embodies knowledge of constraint satisfaction algorithms and other issues pertaining to that domain. Similarly, parser generators such as `yacc` can be seen as implementations of domain specific programming languages for writing parsers.

In relation to parallel programming, domain specific languages offer the hope that the parallel parts of a program can be problem independent (although domain specific) and thus hidden in the implementation of the language so that the user only writes sequential code. For instance, in a domain specific language for event based systems, the user would write event handlers in a sequential language and the system could transparently execute handlers for concurrently occurring events in parallel.

2.2.2 Debugging

The difficulty of debugging multicore programs depends on the programming model. At one extreme, the difficulty of debugging a sequential program that is automatically parallelized is no greater than for a conventional sequential program. At the other end of the spectrum, debugging an explicitly threaded program is complicated by at least three factors:

1. The control state of the program is more complex since each thread has its own point of control.

2. Multithreaded programs are in general nondeterministic, so errors can manifest in one execution and be absent in another (so called Heisenbugs). Needless to say, this complicates testing enormously.

3. Since multithreaded programs in general contain code for synchronization that has no counterpart in sequential programs, that code can exhibit

various problems such as deadlocks that by definition do not occur in sequential programs.

These issues pertain not only to correctness debugging, but also to performance debugging; it is much more difficult to understand how to make a multithreaded program go faster than it is in the case of single threaded programs.

The problems of debugging parallel programs have been attacked by moving to higher level programming models, especially those that have an equivalent sequential reading of the program, and by improved tool support for debugging multithreaded applications.

Tools Tools can attack at least the last two points above by dealing with nondeterminism and by reasoning about or observing the synchronization itself. Low overhead instrumentation for trace collection makes it possible at least to know what happened in a particular execution, and a simulator can give the user precise control over timing, for instance by artificially inflating the time spent in critical sections, making a thread very slow or very fast or simply inserting (pseudo) random delays now and then during execution. If any of these antics provoke an error, the exact same timing can be reproduced to find the source of the error (for instance if a data structure was erroneously overwritten, which lead to a much later memory reference exception).

SICS spinn off Virtutech markets a full system simulator called Simics. Simics simulates the hardware and allows the user to test the entire software stack, including operating system, and provides for repeatable (thus deterministic) timing. Similarly, QuickCheck supports the user in randomized unit testing.

There are also tools that can find some synchronization related errors such as too little synchronization (leading to race conditions in the code) or too much synchronization (leading to deadlock). Race conditions are situations where the result of a program varies unpredictably with the details of thread scheduling and timing. A case in point is the Intel Thread Checker that uses a sophisticated algorithm to find data races. Because of the underlying nondeterminism, the Thread Checker is not guaranteed to find all races, not even all races that are possible with a certain input.

For performance debugging, research at BTH has yielded tools that allow the user to measure and predict parallel performance [3], in particular by profiling the critical path of a multithreaded program which is nontrivial since the critical path potentially moves between threads at synchronization points.

A similar approach is to move away from testing as a validation paradigm towards static verification. Here much work has been done in Uppsala on verifying properties of concurrent programs. While these techniques hold the promise of providing answers that are valid for all possible executions, it is not trivial to scale them to large programs and full programming languages.

Higher level models In task parallel models it is in general possible to run the program sequentially by interpreting task creation as procedure call. This

yields a deterministic sequential semantics of the program. If it can be established that every parallel execution is equivalent to the sequential execution, the parallel debugging problem has been reduced to the sequential one.

This is the approach taken in for instance Cilk, which also provides tool support for run-time checking of the equivalence condition [8]. In practice this condition is related to dependencies between the parallel activities: If no location that is written is accessed by a logically parallel activity, the sequential and parallel executions are equivalent (of course, dependencies involving I/O must also be checked). This tool differs from conventional race checkers in that it is guaranteed to find all errors that are triggered by a particular input.

A similar tool, Embla, has been developed at SICS [7]. It differs from the Cilk tool in not being tied to a specific language. Rather, it works on binaries and is thus largely source language independent. It also differs by working on sequential code and reporting opportunities for parallelization, rather than taking a parallel program and checking whether it is correct.

Similarly, data parallel constructs have a semantics that is independent of the execution order. In this case, the equivalence to sequential execution is built-in.

State of the art There are a number of different tools available for checking properties of explicitly threaded programs, but these are quite slow and their answers are valid only for a particular execution. Formal verification works well for small program (fragments) but has yet to scale to large systems.

Current challenges One major challenge is to scale static techniques to full systems, as that would provide validation of all possible executions.

2.2.3 Programming languages

The programming models that have been proposed have been expressed in a number of different languages and language extensions.

OpenMP The perhaps best known is OpenMP, which is a set of directives added to a sequential base language. Today there are official bindings for C/C++ and Fortran, but implementations for Java exist as well. OpenMP was originally based on a programming model where the worker threads are visible to the programmer. More recently, version 3.0 introduces tasks, and partially reinterprets existing constructs as tasks, but the underlying threads are still visible.

Cilk Cilk is a task parallel extension of C defined at MIT and recently commercialized by the company Cilk Arts as the C++ extension Cilk++. Cilk adds a few keywords to C and every Cilk program has a *C-elision* that is a pure C program formed by removing the Cilk key words. If the Cilk program is deterministic, the semantics of its C-elision (seen as a C program) is identical to

the semantics of the Cilk program. The most important condition for being a deterministic Cilk program is to be free of data dependencies between parallel parts of the program.

X10 X10 is a programming language closely resembling Java that is under development at IBM [4]. X10 aims at supporting parallelism not only at the multicore level, but also across clusters. It has a memory model based on the partitioned global address space model (see Section 2.2.1) so that each computation, object or array element has an associated *place*. A computation can only operate on data in the same place, and computations can fulfill that requirement by spawning computations in arbitrary places.

Erlang Erlang is the result of an effort at Ericsson for developing a language suitable for implementing telecommunications applications. It is based on a dynamically typed, strict functional core extended with processes and primitives for message passing. Each process has its own address space; message passing logically entails copying the contents of the message. For this reason, Erlang is also suitable for programming clusters, but recent implementation efforts have used shared memory to reduce copying, thus making it run more efficiently on multicores.

State of the art Today, most multicore programming is done using either threads (pthreads, Windows threads or Java threads), OpenMP or the Intel TBB.

Current challenges New programming languages generally take quite long to be widely adopted, and when it happens, it is often because of a change in the computing environment. Thus Java adoption was driven by the arrival of the web. Multicore is an even more disruptive technology change, which could drive the adoption of new languages. However, the multicore problem has a large legacy aspect, which was less true of the advent of the web, which might push development in the direction of conservative extensions of existing languages.

2.2.4 Load balancing and scheduling

Scheduling takes place on two different levels in a multicore system. First, the operating system kernel is responsible for scheduling kernel threads on the cores of the processor. Second, for some of the programming models, a user level run-time system schedules more light-weight parallel activities on top of a few heavyweight kernel threads, typically called workers. For the kernel threads programming model, this second layer of scheduling is, if it exists at all, part of the application.

Kernel level scheduling The goals of a kernel level scheduler are fairness, good response time for interactive jobs and good throughput for non interactive

jobs, where a job is a set of threads that cooperate to perform a computation or provide a service. The fairness goal (that all jobs should get a fair share of CPU time) is typically achieved by a combination of time sharing (giving each thread a small amount of CPU time, called a *time slice*, now and then) and space sharing (giving each job a subset of the available cores). While space sharing works at the level of jobs, time sharing works either at the level of individual threads or, if the scheduler always give the threads of a job their time slices at the same time. Of course, a scheduler can employ time and space sharing at the same time.

Kernel level scheduling for multicore processors mainly differ from that of traditional multiprocessors by taking the resource sharing of the cores into account. For instance, a group of cores may be sharing some level in the cache hierarchy, with other groups not sharing. Threads can then either be scheduled on cores in the same group, minimizing communication latency, or spread over several groups, maximizing aggregate cache size to minimize cache misses. Depending on the characteristics of the group of threads, either choice may be preferable. Similarly, with SMT or other forms of multithreaded cores, all functional units are shared so that it might be advantageous to schedule for instance a thread with mainly integer instructions together with a thread with many floating point instructions, in addition to the cache related interactions.

User level scheduling The task and user level threading models are supported by user level schedulers in the run-time systems of the thread/task implementations. For user level threads, the objectives are the same as for kernel threads, but with an expectation of considerably lower cost.

For tasks, the issue of fairness is irrelevant since tasks are non preemptive and can be executed sequentially using a stack. This simplifies the scheduler and contributes to even lower overheads. Task schedulers attempt to simultaneously achieve good load balance (avoiding idle cores), low overhead (avoiding running the scheduler all the time) and good locality (avoiding cache misses when one core needs data computed by another core). These are conflicting goals; from the point of view of locality, the best schedule is typically to run all tasks on a single worker whereas load balancing is best served by spreading the tasks evenly over the machine.

OpenMP schedules loop iterations as tasks in this sense and defines three scheduling policies:

- *static*, which assigns loop iterations to workers before the loop starts, thus minimizing overhead and achieving good locality.
- *dynamic*, where workers obtain loop iterations from a shared counter, optimizing for good load balancing.
- *guided*, which is similar to dynamic but where workers obtain larger chunks of loop iterations in the beginning of the loop and smaller towards the end, as a compromise between the three objectives.

Another class of scheduling algorithms often used for tasks is *work stealing*. Here each worker maintains a local task pool where it pushes and pops tasks in stack like, last in first out (LIFO) order. When a task pool becomes empty, the associated worker attempts to steal tasks from a randomly chosen victim. Typically, the oldest task in the pool is stolen, located at the base of the stack rather than at the top where the victim does its own pushing and popping. This fits well with recursive divide and conquer programs like quicksort where the oldest task in the pool represents as much work as the rest of the tasks in the pool. For programs where the tasks in the pool represent about equal amounts of work, stealing half of the tasks has been proposed.

The dynamic nature of work stealing contributes to good load balancing whereas the stealing of old tasks representing a lot of work gives reasonable locality. If there are significantly more tasks than workers, stealing is infrequent, leading to low overheads.

State of the art Work stealing schedulers are used for example in Cilk and the Intel Thread Building Blocks and OpenMP with its schedulers is widely used.

Current challenges Locality is very important and is strongly affected by scheduling and remains a challenge as discussed in the next section. The best scheduling method also depends heavily on the characteristics of machines and programs, leading to a need for considerable tuning of parallel programs.

2.2.5 Locality

The interaction of local memory usage (cache miss rates), core counts and off-chip bandwidth and latency is likely to be of paramount importance as core counts scale in future multi- and many core processors. As on-chip computational performance increases with increasing core counts, either off-chip bandwidth needs to scale as well or cache miss ratios needs to decrease. If not, congestion will make cache misses slower until they have slowed down on-chip processing speed enough to achieve equilibrium with the limited bandwidth.

Hardware vendors such as Intel are working on meeting the goal of bandwidth scaling, but it will require substantial changes in packaging, typically with memory chips stacked on top of processor chips in the same package, as pioneered in the 80 core Intel Polaris prototype [24, 2]. This architecture gives very short interconnections which helps limit power consumption.

In addition, cache miss rates can be reduced using for instance larger caches, and indeed the total cache size on a multicore chip can easily be made to scale with the number of cores. However, for core counts to scale with density increases, the amount of cache *per core* stays constant. Thus the question becomes whether the total amount of cache can be leveraged to decrease miss rates. This in turn is possible if the code running on the cores share data so that data brought in to the chip to service a cache miss in one core gets reused by other cores before being evicted. This is known as *constructive cache sharing*.

For workloads where different processes are run on the cores (typical of some server environments), there appears to be no straight forward way to reach this goal (except that program text can be shared in an operating system supporting shared libraries). If on the other hand the cores cooperate in running a single application, that application can be written to exploit constructive cache sharing. In the task based style of parallel programming, where a program is divided into a number of tasks much larger than the number of cores, the task scheduler is in a position to exploit constructive cache sharing since it controls which tasks are executed concurrently on the various cores. In a recent study [5], it was shown that this approach can in fact be quite effective.

State of the art The PDF scheduler [5] exploits constructive cache sharing, and the Intel TBB is also moving in the direction of taking locality issues into account [18]. There is quite a lot of work on 3D memory packaging going on, but so far there is no commercial implementation.

Current challenges The work on exploiting constructive cache sharing has just started, so there are many challenges. In particular, there is a trade off between enhancing locality *within* a chip, *between* cores, as the PDF scheduler does, and enhancing locality *within* cores (processors) as traditional work stealing does. In effect, PDF trades an increase in communication between cores for a decrease in off-chip communication.

2.3 Other issues

2.3.1 Transactional memory

In a programming model with explicit concurrent activities (like threads) which share mutable state (for instance shared data structures), it is often the case that operations on these structures are implemented using several memory references that need to be executed without being interleaved with other accesses to the same structure. Incrementing a counter is a simple example; first the old value of the counter is loaded, then the new value is computed and finally the new value is stored in the counter. If a second thread reads the old value between the load and the store, and stores its new value after the sore of the first thread, the update of the first thread is lost; the value of the counter is as if the first thread had not incremented it.

The solution to the problem involves the concept of *mutual exclusion*; while one thread operates on a shared object, no other thread may access it. The standard way to achieve mutual exclusion is to use *locks* which ensure that a thread that attempts to access a shared object while another thread operates on it will be delayed until the operation is completed. A lock can be locked or unlocked; the `lock` operation makes an unlocked lock locked, but applied to an already locked one it waits until the lock is unlocked, then it locks it, while the `unlock` operation simply makes a lock unlocked.

Locks solve the problem of mutual exclusion, but they create problems of their own. For instance, if threads uses multiple locks, as is often necessary,

deadlock may occur. Also, in systems where threads have different priorities, a high priority thread can preempt a lower priority thread holding a lock that the high priority thread itself needs. An intermediate priority thread can then cause the low priority thread to not run, which means that the high priority thread is blocked, effectively waiting for the medium priority thread. This problem is known as *priority inversion*.

In recent years, transactional memory (TM) has emerged as an alternative to locks [9, 20]. In a TM system, operations on shared objects are performed speculatively, without checking if another thread is also accessing it. When the operation is complete, a check is made as to whether another thread accessed the object while the operation was in progress, in which case the operation is *aborted* so that it appears never to have been started. Otherwise it is *committed* and the updates it has performed are made permanent. Transactions are implemented by keeping track of the set of memory location read and written by each transaction and checking that writes in one transaction do not overlap with accesses in another transaction.

Transactional memory can be implemented in hardware as in the original proposal by Herlihy and Moss [9], in software as pioneered by Shavit and Touitou [20] or in some combination [6]. A hardware approach has the best performance, but suffers from a limitation in the size of transactions that can be supported since the set of locations read or written is kept track of in hardware buffers that are of fixed size. Transactions that are too big will always abort. Thus in effect the size of the hardware buffers is visible to the application and becomes part of the ISA. A software implementation keeps the administrative information in memory and even though memory is also of finite size, it is in general "large enough".

State of the art Hardware transactional memory is an active research topic and is implemented in the Rock processor from Sun, due to be available in servers in 2009. A few implementations of software transactional memory is available in prototype form, for instance from Intel [10].

Current challenges The exact semantics of TM systems need to be established, including the interaction with non transactional references. Hardware transactional memory also has its own performance issues [1] that need to be addressed, and its integration with software TM must also be studied.

2.3.2 Thread level speculation

Thread level speculation (TLS) [21, 22]is to the synchronization problem what transactional memory is to the mutual exclusion problem. That is, computations that might be dependent are speculatively executed in parallel and if a dependence violation (the logically earlier computation makes a memory reference that overlaps with one that the logically later computation has already performed) is detected, the logically later computation is aborted and later restarted. If such aborts are infrequent, TLS can achieve good performance.

The main advantages of TLS are that it is applicable in cases where static dependence analysis is unavailable, where the complexity of the code has prevented the analyzer from finding available parallelism or where there sometimes, but not very often, actually exist dependencies (for instance, in just one iteration of an otherwise parallel loop). This disadvantages are the need for hardware support and a relatively narrow zone of applicability: On the one hand, TLS is not needed if static detection of parallelism is successful or the parallelism is explicit. On the other hand, if the parallelism (lack of dependencies) is not there, TLS is not effective. On the third hand, TLS may have a role to play in parallelizing some portions of a program that are not otherwise automatically (or manually) parallelizable, thus mitigating the impact of Amdahl's Law: If a fraction f of the execution time of a program cannot be parallelized, no parallel machine will achieve a speedup that is better than $\frac{1}{f}$.

State of the art No commercial hardware implements TLS and there appears to be no plans in that direction (in contrast to TM). Research has demonstrated a certain potential, but the real size of that potential is unclear.

Current challenges Achieving a scalable implementation is a challenge, as is how to minimize the number of aborts and restarts.

2.3.3 Fault tolerance

As feature sizes shrink (transistors become smaller, wires thinner), it will become more and more difficult to get chips with no defects. Today, DRAM chips are pushing the envelope in device density and are using redundancy to tolerate some manufacturing defects. The same techniques could be applied to multicore processors, and for instance Sun appears to do that already, selling both 8 core and 6 core versions of the UltraSPARC T1 processor with the 6 core version (sometimes) being an 8 core with one or two defective cores.

As core counts increase, we can expect cores to fail dynamically. The question then becomes whether the computation can proceed on fewer cores. Clearly, this depends on the failure mode. A core that starts interacting in random ways with its environment is much more difficult to deal with than one that just stops interacting which is again more difficult that dealing with one that signals its ill health explicitly (for instance because it has started to accumulate parity errors).

On a chip with many cores, redundancy could be used to achieve a high degree of tolerance, at least for the last category above.

Also, process variations are likely to make the maximum clock frequency of cores vary, an effect that must be taken into account when scheduling. This effect would favor dynamic (on-line) scheduling algorithms over static (off-line) ones.

State of the art Defective cores are sometimes disabled at manufacture, but no current multicore processor can continue executing if a core ceases to function properly at run-time.

Current challenges Dealing with run-time failures is a big challenge, especially in a cache coherent system. Performance variations in cores due to manufacturing and temperature variations must also be dealt with as they complicate load balancing.

3 Swedish multicore related activities

This section presents some of the work that is ongoing in Sweden, both from an academic and industrial perspective.

3.1 The Swedish Multicore Initiative

The Swedish Multicore Initiative is a concerted effort to address the engineering and strategic issues related to multicore processor technology for the software intensive systems industry in Sweden. The Initiative ties together all parties interested in advancing this technology with the main objective of drastically reducing the cost of software production for multicores.

The vision of the Initiative is to make multi/many-core microprocessor technology as easy to use for the Swedish software intensive industry as single-core microprocessors.

The main objectives of the Initiative therefore include:

- To make Swedish software-intensive industry internationally competitive in utilizing multi/many-core technology
- To make graduates from Swedish universities internationally competitive in utilizing multi/many-core technology
- To make Swedish research internationally competitive in advancing state-of-the-art in utilizing multi/many-core technology

Our belief is that this can only be achieved through a focused collaboration between industrial and academic organizations. To facilitate this, the Initiative will form a virtual center which acts as a one-stop shop for competence in utilizing multi/many-core technology. This center could be seen as a Swedish counterpart to international industrial/academic partnerships such as the one at UC Berkeley and University of Illinois UC (with Microsoft, Intel US) and at Stanford University (with AMD, HP, Intel, NVidia and Sun). The center naturally connects to international competence networks through its members. One example is the strong link to the HiPEAC Network of Excellence supported by EU under FP7.

3.1.1 Activities

The Swedish Multicore Initiative has a number of activities to meet its objectives:

- Dissemination of research results and best practices:
  - Multicore day: A state-of-the-art annual seminar for industry and academia highlighting technology advances, research results and hands-on solutions to current problems. It will be held in September yearly.
  - Swedish Multicore Workshop: An annual workshop for academia and industry to present and discuss recent research results. The first workshop will be organized by Blekinge Institute of Technology in November 2008.
  - Best practices workshop: An annual workshop for industry and academia to present and discuss best practices in multicore software development. First BP workshop will be arranged in February 2009.
- Research and educational program to set the agenda for research and educational efforts in multi/many-core technology from a Swedish perspective. Working groups will be formed to initially focus on
  - A technology roadmap from a Swedish industrial perspective
  - Curriculum development
  - Coordination and marketing of Swedish multicore competence
- Collaborative research between academic and industrial groups.

3.2 Academic work in Sweden

In this section we discuss academic work on multicore related issues in a roughly north to south order.

3.2.1 Uppsala university

At the Department of Information Technology, Uppsala University, the UPMARC center has recently been formed to make a coordinated attack on the challenges of developing methods and tools to support software development for multicore platforms. UPMARC brings together research groups in complementary areas of computer science: computer architecture, computer networks, parallel scientific computing, programming language technology, real-time and embedded systems, program verification and testing, and modeling of concurrent computation. Research directions span over programming language constructs, program analysis and optimization, resource management for performance and predictability, verification and testing, and parallel algorithm construction and implementation. UPMARC has recently been awarded a ten year Linnaeus grant from the Swedish Research Council, as a witness of scientific excellence. This funding is a very good foundation for performing basic research, which should be complemented by more applied research efforts, in collaboration with industrial and scientific applications in multicore computing. We are actively

building up such collaborations, and any funding for these efforts is welcome and can take advantage of existing research activities.

Research in UPMARC will use the development of a number of concrete parallel software bases as drivers for research and test-beds for ideas. We plan to use applications in high-performance computing (e.g., climate simulation), in mobile phones (e.g., protocol processing), in programming language implementations (e.g., runtime system implementations), in embedded control applications (e.g., target tracking or robot control), and also other areas. These efforts per se can not be funded by the UPMARC grant, but we seek collaboration schemes to realize them.

The planned research in UPMARC is structured into a number of research directions. We have a very strong track record in each of them, and will use our expertise to address challenges for multicore software development. developing principles for algorithm construction in key application areas, considering the new trade-offs for multicores in comparison with previous multi-computers.

- In the scientific and high-performance computing community, parallel programming and parallel algorithms have been central tools for more than twenty years. With multicore platforms becoming mainstream, many applications, if not most, need to be adapted. This includes systems software as well, i.e. operating systems, communication subsystem and execution environments for parallel languages. We have been working on parallel algorithms and programs for scientific applications and communication systems for the last 20 years. Leveraging on our expertise we will focus on the following long term challenges:
  - New scientific applications that scale up to a large number of cores, i.e. hundreds or thousands of cores.
  - How to design protocols and communication algorithms for multicores.
- Developing techniques for making the most efficient use of system resources, including processor cores, memory units, communication bandwidth, in order to meet requirements of performance and predictability. We will develop techniques by which the wide variety of resources can be abstracted, modeled, managed and analyzed. Our research will focus on two challenges.
  - Efficient management of shared resources for performance: we will develop techniques for modeling resources, as a basis for identifying bottleneck, code transformation, and self-adapting run-time resource allocation techniques.
  - Predictability of timing and resource-consumption: we will develop techniques to predict bounds on timing and resource usage e.g. energy-consumption.

- Developing programming language constructions and paradigms, that allow the software developer to express the potential parallelism of an algorithm, at the same time as shielding her from the added complexity of concurrency. Here, we plan to continue our work on the efficient implementation of Erlang-style concurrency on multicores, and investigate how message-passing concurrency compares to and can be combined with atomicity constructs. We will develop annotations and contracts, which allow programmers to specify properties of software components at a significantly higher level then is currently possible, and accompanying program analysis and testing techniques for checking these annotations. We will also build a framework for formulating and proving correctness of optimizing transformations. A long-term goal is to build a library of transformations along with their formally machine-checked correctness criteria.
- Developing techniques for analyzing vital correctness properties of concurrent programs, by developing and combining techniques in formal verification, static analysis, and testing.

3.2.2 Mälardalen university

The Multicore research at Mälardalen University is mainly carried out by the Programming Languages group[2], with some planned activities in the Real-Time Systems Design group[3]. These are the main planned activities:

Parallelization of legacy telecom software This is a topic of great interest for Swedish telecom industry, where many millions of lines of code are invested in the current, mainly single-core systems. Automatic parallelization of this code, to make it run on multicore processors, would relieve the industry from the huge effort of rewriting the code by hand. Automatic parallelization is very hard in general, but telecom code has certain characteristics that may make the problem more tractable. We have previously studied the problem in a project with Ericsson, and some possible parallelization methods have been suggested. We want to continue this research, but would then need further funding. Such funding has been sought from SSF, together with BTH, Chalmers, SICS, Ericsson, and Enea.

WCET Analysis for Multicore and MPSoC systems Worst-Case Execution Time (WCET) analysis finds upper bounds for the largest possible execution time of a piece of code on a certain hardware. This information is crucial when verifying the timing properties of safety-critical real-time systems. Such systems are found in applications such as automotive, and WCET analysis is thus highly relevant to Swedish industry. The Programming Languages group is one of the world-leading groups in this area. Current WCET analysis methods

[2]http://www.mrtc.mdh.se/index.php?choice=research_groups&id=0009
[3]http://www.mrtc.mdh.se/index.php?choice=research_groups&id=0006

and tools, as well as almost all scientific literature in the area, concern exclusively single-core systems. The introduction of multiple-core systems changes the rules of the game completely. Scientifically, WCET analysis for multicore systems is almost unchartered territory. However, it not hard to see that timing predictability of code running in such systems can be drastically reduced, due to unpredictable access times to shared resources such as buses and shared memories. Within the EU FP7 NoE ARTIST-DESIGN on embedded systems design there is a Timing Analysis activity, lead by the MDH group, whose purpose is to initiate research in this area. This research will have to be cross-disciplinary and preferably involve also researchers in computer architecture and system design. The NoE only supports networking activities: funding to do the actual research must be sought elsewhere, for instance nationally.

Real-Time Scheduling for Multicore Systems The Real-Time Systems Design group performs research on different aspect of real-time scheduling methods. They are now moving into the area of real-time scheduling for multicore systems.

3.2.3 Royal Institute of Technology and Swedish Institute of Computer Science

Royal Institute of Technology (KTH) and SICS have a joint research group in multicore technology. The work focus on programming models and support for resource management on manycore processors.

Programming models for manycore processors Manycore (more than 10-20 cores) processors require a radically different mindset than what is mostly used on today's multicore processors. With just a few cores, it is hard, but still feasible to reason about threads of control and their interaction. With manycores, this is no longer feasible. We advocate the use of *safe task-based parallelism*. With this model programmers reason about *tasks*, small pieces of code that may be executed independently of other tasks given that data dependencies are still observed. In a *safe* program, there are no dependencies between concurrently executing tasks. All execution orders (schedules) of a safe task parallel program have the same semantics, including a canonical sequential execution. Thus all program development and correctness debugging can be done in the sequential domain while performance debugging is done in the parallel domain.

Our work involves tools for analyzing and exposing data dependences and efficient implementations of task-based parallelism.

- Embla is a dynamic data dependence analyzer (profiler) that can be used to discover opportunities for parallel execution in sequential programs. It is based on the Valgrind instrumentation infrastructure and is independent of the source language of the analyzed program. Since Embla is used with a sequential program, the results are safe with respect to the inputs used for the analysis run, in contrast to tools for explicitly parallel programs.

Thus Embla supports the development of task parallel programs that are safe by construction.

- Wool is a simple implementation of task scheduling which differs from other widely used alternatives like OpenMP, Cilk or TBB by requiring no compiler support and having a simple direct style C-based API.

Resource management on manycore processors Manycore processors contain numerous resources that must be managed in run-time robustly and efficiently. In general applications, the workload of processors will vary greatly over time. Typically, the workload will consists of bursts of self-similar nature. In such systems, it is important to control the hardware so that enough resources are available for the workload, but not more, for energy-savings reasons. In previous work we have integrated simple periodic shutdown strategies in a commercial-grade operating system with the purpose of switching cores of a small-scale multicore processor (8 cores) on and off to adapt to the current workload needs with up to 80% energy savings as a result.

This work will continue taking many more resource variabilities and fault-tolerance into account.

3.2.4 Linköping university

Optimized On-Chip Pipelining of Memory-Intensive Computations on Cell BE Memory-intensive computations, such as stream-based sorting or data-parallel operations on large vectors, cannot utilize the full computational power of modern multi-core processors such as Cell BE because the limited bandwidth to off-chip main memory constitutes a performance bottleneck. We apply on-chip pipelining to reduce the memory transfer volume, and develop algorithms for mapping task graphs of memory-intensive computations to Cell that also minimize on-chip buffer requirements and communication overheads. (Contact: C. Kessler)

Context-aware composition of parallel programs from components Programming parallel systems is difficult. Components are a well-proven concept to manage design and implementation complexity, but are often more general than necessary and hide too many design decisions such as scheduling or algorithm selection, which should better be bound later (e.g. at run-time) when more information about available resources or problem sizes is known. We investigate context-aware composition, a powerful optimization technique that can be seen as a generalization of current auto tuning methods for domain-specific library functions. (Contact: C. Kessler)

High-level parallel programming for Cell BE Exploiting the full performance potential of heterogeneous multi-core processors such as Cell BE is difficult, as several sources of parallelism (inter-core, SIMD, and DMA parallelism) must be coordinated explicitly and at a low level of abstraction. We apply the

skeleton programming approach to reduce this complexity to, basically, that of ordinary sequential programming wherever the computation structure fits a specific pattern for which generic parallel code parameterizable in problem-specific code is available. Our implementation BlockLib for Cell BE achieves the efficiency of hand-tuned code. (Contact: C. Kessler)

Low-latency, low-power and low-silicon-cost architecture for predictable real-time computing So far, there is no perfect architectural solution for low-latency, low-power, and low-silicon-cost real-time computing. Requirements for such applications tend to go beyond the capacity that current silicon technology and classical general-purpose computer architecture can offer. An example is the data recovery signal processing for long-term evolution (LTE). Many real-time computing applications are predictable, which allows to use template-based, domain-specific programming models and tools. We develop a reconfigurable multi-core architecture and programming tool chain aimed at accelerating multimedia, telecommunication and other vector computing applications. (Contact: D. Liu)

Automatic parallelization of mathematical models Equation-based object-oriented mathematical modeling languages such as Modelica allow to express simulation problems at a high level of abstraction. We extract parallelism over time steps and over the system directly from the high-level model, and also exploit parallelism in the numerical methods used for simulating models. This includes task graph mapping, pipelining and replication techniques. The results are also applicable to single-assignment programs in general. Our implementation in the OpenModelica open source compiler achieves a speedup of 6.1 on 8 cores for a flexible-shaft mechanical model. Code generation for Cell BE is under way. (Contact: P. Fritzson)

Large-scale parallel simulation of mechanical applications on multi-core clusters Research on accelerating mechanical system simulations by parallelization is a long-term cooperation between SKF (Göteborg) and Linköping University. Current emphasis is on detailed modeling and simulation of contact problems on large multi-core clusters. Current research problems include improved hybrid scheduling and increasing the amount of parallelism in simulations. (Contact: D. Fritzson, P. Fritzson)

3.2.5 Chalmers Multi-core Initiative

Software development productivity is a first-class citizen. Unfortunately, writing software to take advantage of multicore computers is a difficult, time-consuming, and error-prone process that will cut down software development productivity by orders of magnitudes. While there is an educational gap to fill, which we as well as all other universities are addressing, this does not solve the problem in the long-term. What is needed is to encapsulate multicore computers in a

way so that they expose a clean abstraction that a software developer is used to. The missing link is what this abstraction should be and how it should be implemented. This is the key research questions that we at Chalmers are committed to address.

The Chalmers Multicore Initiative is committed to do research on a productive abstraction that enables software developers to focus on delivering functionality and delegates how to exploit the performance of multicores to underlying libraries and architectural abstractions. Our approach to do that builds upon our competences in

- Programming language and program analysis (Sheeran, Hughes, Claessen)
- Parallel algorithms and shared data structures (Papatriantafilou, Tsigas)
- Computer Graphics (Assarsson)
- Computer architecture (Stenstrom, McKee)

At the core abstraction exposed to the software developer, we are researching domain-specific language implementations and associated program testing methodologies that abstract away architectural details to developers. Core values directly addressed are enhanced software development productivity. To implement such abstractions, it is important to leverage libraries and system software components that can be proven to execute correctly in a multicore environment. Our long-term research on lock-free (synchronization-free) data structures is leveraged here. Finally, the basic architecture of a multicore must expose a more productive interface to the software. Our current focus on transactional memory is key to such a direction.

3.2.6 Blekinge Tekniska Högskola

The Parallel Architectures and Applications for Real-Time Systems (PAARTS) research group, http://www.bth.se/tek/PAARTS/, at BTH has almost 20 years of experience of parallel applications, multiprocessors, and multicore systems. Several of the research projects have been done in close cooperation with industry. Examples of industrial partners are Ericsson AB, UIQ Technology AB, Vodafone (now Telenor) AB, Danaher Motion Särö AB, and Sony-Ericsson AB. The PAARTS group currently consists of three full professors, four assistant/associate professors, three lecturers, and four PhD students. The group is part of the BESQ Research Center, http://www.bth.se/besq/, which consists of five full professors, 18 assistant/associate professors, and 24 PhD students.

The PAARTS research group is mainly working on the following topics:

- Methods, guidelines, and tools for cost-effective development of parallel applications for multiprocessors and multicore systems. This includes techniques for handling conflicts and trade offs between performance, maintainability, and availability.

- Tools and methods for visualization, performance prediction, and performance debugging of multithreaded applications. In this area a prototype tool has been developed that visualizes the parallel execution of a multithreaded program, shows all synchronizations and thread interactions, and predicts the performance for an arbitrary number of processors.
- Methods and techniques for migrating and porting of sequential applications to multiprocessor and multicore systems. We have worked with both user-level application as well as kernel-level real-time operating system code.
- Parallel computer architectures and multicore architectures, with a focus on the cache and memory system. For example, novel coherence protocols and update strategies have been developed. Current interests are novel synchronization and data management techniques such as transactional memory.
- NP-hard resource allocation algorithms, e.g. scheduling. In this area focus is on mathematical techniques for establishing optimal performance bounds on NP-hard resource allocation problems, thus making it possible to compare the performance of heuristic resource allocation techniques with the optimal result.
- Real-time multimedia applications. In this area the focus is on techniques for parallel real-time ray tracing of dynamic scenes and real-time streaming video applications in mobile terminals with various resource constraints such as power and bandwidth.

3.2.7 Halmstad University

Within CERES, the Centre for Research on Embedded Systems at Halmstad University, research on embedded highly parallel computing for high-performance applications has been going on for twenty years. The research is and has been performed in close collaboration with Swedish industry developing advanced telecommunication and radar system equipment (i.e., Ericsson and Saab). Collaboration also takes or has taken place with parallel processor developers such as Ambric, XPP and Clearspeed and with academic institutions such as UC Berkeley and MIT.

Currently, the main activities are within the project EPC – Embedded Parallel Computing, with Combitech, Ericsson and Saab Microwave Systems as industrial partners. Two of the main tasks in the project are the following:

A Domain-specific Approach for Software Development on Manycore Platforms For complex real-time systems, such as baseband processing platforms, we see a need for tunable code parallelization- and mapping tools, allowing programmers to take the system's real-time properties into account during the optimization process. Therefore, complementary to fully automatized multicore compilers, we are proposing an iterative code parallelization- and mapping tool

that allows the programmer to tune mapping by: analyzing the result of a parallel code map using performance feedback giving timing constraints, clustering and core allocation directives as input the tool Such a tool would allow the programmer early in the development process to explore the run time behavior of the system and to find successively better mappings. We believe that this iterative, machine assisted workflow helps keeping the application software portable and supports the user to make trade-offs concerning throughput, latency and compliance with real-time constraints on different platforms. The tool is based on well defined dataflow models of computation for modeling applications and manycore targets, as well as the base for our intermediate representation for manycore code-generation.

High-Level Programming of Coarse-Grained Reconfigurable Architectures Reconfigurable computing devices have evolved over the years from gate-level arrays to a more coarse-grained composition of functional blocks or even program controlled processing elements. From a programming perspective, the programming methods are moving from low-level structural descriptions to high level languages. We believe that coarse-grained reconfigurable processor arrays are well suited to cope with the increasing demands of high-performance embedded systems because of their ability to adapt according to the application requirements. Being basically reconfigurable, these architectures offer the possibility to modify their structure for supporting new functionality in the future. From the programming perspective, the programming models are to be based on computation models with the inherent semantics for separating computations from communication. A promising approach is to use programming models with the abstractions of processes to express concurrency and logical channels for specifying communication, in order to describe the functionality of the arrays of processors. Based on the findings, we intend to perform some experiments by adopting selected well-known computation models to program some example coarse-grained architectures. In particular, we currently work with the CSP- and pi-calculus based language occam-pi in order to produce code for the Ambric processor array.

Involved researchers: Bertil Svensson, Verónica Gaspes

Involved PhD students: Jerker Bengtsson, Zain-ul-Abdin

3.3 Industrial work

The Swedish system industry is in general software intensive and as such a receiver of multicore work. In this report, we will not look in detail on their specific challenges. However, there are also suppliers of multicore related technologies, some of which we will briefly discuss here.

3.3.1 Nema Labs AB

Nema Labs (www.nemalabs.com) mission is to offer tools that allow software developers to concentrate on delivering functionality and to leave the task of reliably threading of legacy and new code to leverage on the performance of

multicores to intelligent tools. While tools and methods to parallelize C/C++ code are available, they either reduce software development productivity by offering a too low level of abstraction (e.g by using OpenMP) or by not leveraging the performance potential of multicores (auto-parallelizers). By contrast, Nema Labs roadmap realizes tools that allow software developers to thread legacy C/C++ code fast and reliably by using a sequential abstraction with the help of automatic threading in an iterative fashion.

Nema Labs is a spin out of research at Chalmers University of Technology. It is offering a roadmap of threading tools that innovates on top of automated C/C++ source code threading products to accelerate reliable threading of legacy code. The first product, called FASThread (Fast Automated Software Threader), will be released in early 2009 as a Linux plug-in for the Eclipse Integrated Development Environment (IDE), and will provide developers a familiar setting for threading their C programs with a few simple, automated steps. Shortly thereafter, plug-ins for other IDEs and operating systems will be released followed by a C++ version as well.

3.3.2 Acumem AB

The move to multicore architectures often results in lower-than-expected performance. Limited per-thread cache capacity, memory bandwidth bottlenecks and inefficient thread interactions ruins the potential performance, but only a handful of experts are capable of understanding and fixing such problems.

This lead to the creation of the start-up company Acumem, partly based on the work by Professor Erik Hagersten and his research team at Uppsala University. Many years ago they saw the need for tools that enable non-experts to work efficiently with multicores, but it can be equally important to provide tools to make the performance experts more productive, since they can be expected to get their hands full with performance problems over the next decade.

One part of Acumem's core technology is a sampler that efficiently captures sparse architecturally-independent runtime information from native execution – the application's fingerprint. A second part is a modeling technology that models a given multicore's behavior based on a fingerprint. These two parts together gives an unprecedented insight into what really goes on inside a multicore with high accuracy and high efficiency.

Acumem has developed their first product, SlowSpotter^{TM}, on top of the core technology. It finds application SlowSpots^{TM} – places in the code that can be modified to improve the application's performance. About 20 different types of SlowSpots^{TM}, related to multithreaded execution and cache usage, are identified and fixes suggested at a level of detail allowing for non-experts to perform the optimization. A second product SlowSpotter-Pro^{TM} is targeting the performance experts with an aim at improving their productivity. The freely available SpotLite^{TM} can tell if an application has SlowSpots^{TM} and classifies them according to type.

The Acumem tools are included in HP's multicore toolkit and AMD's ans Sun's recommended 3rd party performance tools. Other close partners include

Microsoft, Intel and SGI.

4 Glossary

This section provides a glossary of terms used in this report, most of them related to multicore. It should not be taken as exhaustive.

anti dependence A dependence between a read of a memory location or register and a subsequent write to the same memory location or register. Causes serialization of the accesses to avoid unintended overwrite before read of the old value. Can be eliminated by renaming (see *register renaming*).

architecture (also *instruction set (architecture), ISA*)

cache A smaller faster memory containing a subset of the information stored in a larger slower memory. Information read from the larger memory is placed in the cache so that subsequent references can be serviced more quickly.

cache hit A cache access where the referenced address currently is in the cache.

cache miss A cache access where the referenced address is currently not in the cache.

core (also *processor core*) The fundamental building block of a multicore processor, consisting of functional units, control logic, one or more register sets and first level caches for data and instructions. Sometimes also a second level cache is included in each core.

DRAM Slow, high capacity memory with read latency of a few tens of nanoseconds. Used to build the main memory of current computers.

dual core processor A processor with two cores.

first level cache (also *L1-cache*) The cache closest to the processor in a multilevel cache hierarchy.

flow dependence A dependence between a write to a memory location or register and a subsequent read of the same memory location or register. Causes serialization of the accesses to ensure that the correct value is read.

functional unit A hardware circuit realizing a limited set of related operations, for instance arithmetic and logical operations, floating point arithmetic, or shifts.

multilevel cache hierarchy A set of caches organized so that the smallest (and fastest) cache is accessed first, and if the access misses, the next larger cache is accessed and so on. Typical processors today have two levels of cache, but three level caches are found in some processors such as the AMD Agena quadcore processor.

out of order issue In an out-of-order issue processor, the instructions do not necessarily execute in the order they have in the program. Instead, when they have been fetched from memory and decoded they are placed in a buffer in the processor (called an *instruction window*) and executed when their operands are available, increasing the amount of instruction level parallelism the processor can exploit. A modern out-of-order core can have on the order of a hundred instructions in its instruction window.

output dependence A dependence between two writes to the same memory location or register. Causes serialization of the accesses to ensure that the correct value is stored for later reads. Can be eliminated by renaming (see *register renaming*).

quad core processor A processor with four cores.

register A small fast memory structure capable of storing one word (currently typically 64 bits) in a processor core.

register file A hardware structure containing a collection of registers, at least as many as specified by the architecture but often more than that in processors implementing register renaming.

register renaming An implementation technique for allowing parallel execution of several instructions that use the same register for storing different values by directing writes to the same (logical) register by different instructions to different physical registers. Eliminates anti and output dependencies on registers.

second level cache (also L2-cache) The second closest cache to the processor (with the first level being the closest). Misses in the first level cache are serviced by the second level cache.

superscalar A superscalar processor executes instructions that appear close to each other in the instruction stream in parallel. Most of today's processors are superscalar.

task A unit of parallel execution in some programming models. Different tasks in the same program typically share data, but synchronization is restricted so that it does not prohibit a simple stack based sequential execution order; a simple example of how to achieve this result is to only synchronize at task creation and termination. In some contexts, the term task is used to refer to a unit of parallel execution with arbitrary synchronization, although this report uses the term *thread* for this purpose.

thread A unit of concurrency in some programming models. Different threads in the same program typically share data and use explicit synchronization to coordinate their activities.

VLIW A *very long instruction word* machine an instruction set where each instruction specifies a number of different operations to be performed in parallel, each with their own operands. The programmer (or compiler) is responsible for ensuring that the operations are independent.

worker In programming models with light weight parallelism, such as the task parallel model, the light weight parallel activities are scheduled on top of a few (typically one per core or hardware thread) heavyweight threads called workers by the parallel run-time system.

This concludes the glossary.

References

[1] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. *SIGARCH Comput. Archit. News*, 35(2):81–91, 2007.

[2] Shekhar Borkar. Thousand core chips: a technology perspective. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 746–749, New York, NY, USA, 2007. ACM.

[3] M. Broberg, L. Lundberg, and H. Grahn. VPPB: A visualization and performance prediction tool for multithreaded Solaris programs. In *International Parallel Processing Symposium*, pages 770–776, 1998.

[4] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[5] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, New York, NY, USA, 2007. ACM.

[6] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, 2006.

[7] Karl-Filip Faxén, Konstantin Popov, Sverker Jansson, and Lars Albertsson. Embla—data dependence profiling for parallel programming. In Fatos Xafa and Leonard Barolli, editors, *CISIS 2008: Proceedings of the Second*

International Conference on Complex, Intelligent and Software Intensive Systems, pages 780–785. IEEE Computer Society, 2008.

[8] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 1–11, New York, NY, USA, 1997. ACM.

[9] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[10] October 2008. software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20.

[11] Tim Johnson and Umesh Nawathe. An 8-core, 64-thread, 64-bit power efficient sparc soc (niagara2). In *ISPD '07: Proceedings of the 2007 international symposium on Physical design*, pages 2–2, New York, NY, USA, 2007. ACM.

[12] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 272–282, New York, NY, USA, 1989. ACM.

[13] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[14] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. Ibm power6 microarchitecture. *IBM J. Res. Dev.*, 51(6):639–662, 2007.

[15] Gabriel H. Loh. 3d-stacked memory architectures for multi-core processors. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 453–464, Washington, DC, USA, 2008. IEEE Computer Society.

[16] Mark Moir, Kevin Moore, and Dan Nussbaum. The adaptive transactional memory test platform: a tool for experimenting with transactional code for Rock (poster). In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 362–362, New York, NY, USA, 2008. ACM.

[17] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural*

support for programming languages and operating systems, pages 2–11, New York, NY, USA, 1996. ACM.

[18] A Robison, M Voss, and A Kukanov. Optimization via reflection on work stealing in TBB. In *Proc. of Parallel and Distributed Processing*, 2008.

[19] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.

[20] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[21] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, New York, NY, USA, 1995. ACM.

[22] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[23] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, New York, NY, USA, 1995. ACM.

[24] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, A. Singh, T. Jacob, A10, S. Jain, A11, S. Venkataraman, A12, Y. Hoskote, A13, N. Borkar, and A14. An 80-tile 1.28tflops network-on-chip in 65nm cmos. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 98–589, 2007.

[25] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.