Fine-Grained Multithreading on the Cray T3E

Andreas Grävinghoff and Jörg Keller

FernUniversität-GHS Hagen, FB Informatik, D-58084 Hagen, Germany
`Andreas.Graevinghoff|Joerg.Keller@FernUni-Hagen.de`

Abstract. Fine-grained multithreading can be used to hide long-latency operations encountered in parallel computers during remote memory access. Instead of using special processor hardware, the emulation of fine-grained multithreading on standard processor hardware is investigated. While emulation of coarse-grained multithreading is common in modern operating systems, in the fine-grained case research on emulation has been limited and design of multithreaded processors has been favored. A set of tools was developed to support emulation of multithreading on the Cray T3E parallel computer. Several experiments based on parallel matrix multiplication were performed.

1 Introduction

An important problem faced by parallel computers is the latency of accessing remote memory. As this latency usually increases with the number of processors, massively parallel computers are especially affected. While the latency of remote stores can be ignored since they return no result, remote loads have to be completed before computation can proceed (at least beyond a certain point).

A popular approach to attack this problem is to avoid latency by the use of coherent caches. However, this approach causes widely varying memory access times, therefore these machines are called ccNUMA (cache coherent non-uniform memory access) architectures. The non-uniform access time complicates application development, especially for irregular applications. Instead of avoiding latency, one can try to hide latency by *multithreading*, which is explained in the next paragraph.

A multithreaded processor switches context between different *threads* in order to perform useful computations while other threads are waiting for completion of outstanding operations, thus hiding the latency of those operations from the user. This is only possible if several threads per processor are available, hence the application must possess more parallelism than in a ccNUMA machine of comparable size. However, many applications possess this amount of parallelism if the number of threads per processor is moderate.

We distinguish between *fine-grained* (switches context after one or a few instructions) and *coarse-grained* (switches context after a block of instructions) multithreading. In the context of parallel computers, fine-grained multithreading is more interesting (e.g. to achieve higher degrees of parallelization). Multithreading can be implemented by special processor hardware or by emulation in software on off-the-shelf hardware. Special processor hardware is expensive, time-consuming to design, error-prone and often slower than commercial processors. Examples for multithreaded processors are Sparcle [AKK$^+$93], Anaconda [Moo96], SB-PRAM [KPS94] and the Tera MTA [BH95]. Emulation in software is state-of-the-art in coarse-grained multithreading: it is called *multitasking* and is used by almost every modern operating system. Several new operating systems (e.g. Solaris) support threads in the form of *lightweight processes.* The purpose of these lightweight processes is not latency hiding but reduced overhead compared to normal processes and abstraction from the number of processors available. However, the overhead is still too high for our purposes, hence these lightweight processes are not considered further.

We will describe the basic concept in the following section. The third section covers the implementation on a Cray T3E in detail, along with a description of the developed tools. Experimental results based on a parallel matrix multiplication algorithm are presented in the fourth section. An outlook with special emphasis of ongoing and future work concludes the paper.

2 Basic Concept

Emulating fine-grained multithreading on off-the-shelf processors can be done as follows: For each thread, the executed program as well as the *context* (processor state of the thread) are stored in memory. The data structure that contains the context of a thread (and some management information for the emulation program), is called *frame.* To execute an instruction from a given thread, the emulation program restores the processor state, fetches and executes the instruction and updates the processor state in memory. Afterwards, the next thread is executed. To decrease the time to switch contexts, only the part of the context used or modified by the fetched instruction is restored or saved. Note that we assume the emulating and emulated instruction sets to be identical. If this is not the case, then the emulated instructions have to be replaced by one or more instructions of the emulating instruction set.

```
                                ldq  r1, #rs1(frame)
                                ldq  r2, #rs2(frame)
<op> rs1, rs2, rs3     ──→      <op> r1, r2, r3
                                stq  r3, #rd(frame)
                                ret
```

Fig. 1. Example for Subroutine Generation

To implement emulation of mutlithreading, we modify the program code to replace every instruction block `I` by a subroutine. These subroutines restore every register that will be read or modified by the instructions in `I`, execute the instructions in `I`, save every register that has been written or modified, and return. An example of the modification process for an instruction block containing a single instruction `op` is shown in Fig. 1. The left side contains the original instruction `op`, which uses two registers `rs1, rs2` as operands and stores the result in register `rs3`. The corresponding subroutine is shown on the right side: First, the registers `r1, r2` are loaded with the values of `rs1, rs2` from the current frame. Afterwards the instruction `op` is executed, storing the result in a third register `r3`. This register is stored to the current frame before the subroutine returns. Note that the registers `r1`, `r2`, `r3` are not necessarily identical to registers `rs1`, `rs2`, `rs3`, but can be any of the architected registers.

The size of the instruction block can be as small as a single instruction, thus enabling very fine-grained multithreading. The overhead associated with emulation of multithreading decreases with increasing block size since fewer context switches are required, but large block sizes may complicate latency hiding. Note that the size of the instruction block is variable, i.e. it is not necessary to use the same blocksize throughout the whole program. This fact makes further optimizations possible, e.g. identifying sequences of instructions that operate on the same registers and omitting unneccessary accesses to the frame.

The number and location of registers to be read, written or modified by instructions in `I` can be determined from the instruction set architecture and the instruction itself. This information is used during generation of the corresponding subroutine. The emulation program basically consists of a single loop that calls these subroutines from all threads in a round-robin manner:

1. Initialization
2. Load program counter (PC) from current frame into register *threadPC*.
3. Execute subroutine by jump to value stored in *threadPC*, saving old program counter in register *mainPC*.
4. Return from subroutine to program counter stored in *mainPC*, saving old program counter in register *threadPC*.
5. Store program counter from register *threadPC* to current frame.
6. Load pointer to next frame into register *framePtr*.
7. If the next thread is not the last one, go to step 2.

If synchronous operation of multiple processors is desired, a barrier synchronisation can be performed after each round, i.e. after each subroutine from every thread on a given processor has been executed. The main loop, which uses three different registers (*threadPC*, *mainPC*, *framePtr*), should be compact and fast, since execution is required for every subroutine. We will cover these details in the following section.

We assume that the high-level language source code of the programs to be emulated is available and that the program already uses threads. The thread-related system calls (e.g. creation, deletion) are then replaced by our own routines during recompilation. After modification of the assembler source the program is linked with a small library containing our routines (e.g. main loop). We further assume that all threads operate in user mode. Thus only the registers accessible in user mode are shared between different threads and form the context of a thread. By confining threads to user mode, the switching time between threads is significantly reduced. Calls to the operating system are not emulated, but are executed as usual by system calls and traps, i.e. there are no changes to the operating system. Obviously, we can not handle self-modifying code, since we perform all code modifications during compilation. However, this is not a serious restriction since self-modifying code is generally considered as unfavorable.

3 Implementation

Emulation of Multithreading was first implemented on the Alpha architecture. Alpha is a 64 bit architecture, i.e. all registers are 64 bits wide, instructions are 32 bits wide. There are 32 integer as well as 32 floating-point registers, one register in each class is a dedicated zero source/sink register. Apart from the floating-point control register `FPCR`, which is only used for some rounding modes, there are no special registers. Alpha is load/store architecture, i.e. all operations are performed between registers. Supported data types include longword (32 bit) and quadword (64 bit) integers as well as five different floating-point formats (2 VAX, 3 IEEE). Recently, support for byte (8 bit) and word (16 bit) integers as well as motional video instructions were added to the Alpha architecture. Implementations of the Alpha architecture, e.g. the 21064, 21164 and 21264 chips, continue to score top ratings on the SPEC benchmarks since the introduction of the architecture in 1992.

The Alpha architecture was chosen for several reasons: Since there are no special registers or condition codes, the user moce context contains only data registers, the program counter (`PC`) and the floating-point control register (`FPCR`). The `FPCR` has to be saved/restored only for some floating-point instructions, otherwise only the general-purpose registers specified in the instruction have to be saved/restored, which eases the work of the assembler converter. Manipulation of the program counter can be done easily with the supported jump-type instructions, which makes implementation of the main loop straight-forward. Last but not least, massively parallel computers based on the Alpha architecture are available, e.g. the Cray T3E, which is described in the next subsection.

3.1 Platform

The Cray T3E supports between 2 and 2048 processing nodes interconnected by a bidirectional 3D torus. Each processing node contains a 21164 Alpha processor running at 300, 450 or 600 MHz. The 21164 is a superscalar processor featuring two integer as well as two floating-point function units and a sustained (in-order) issue rate of four instructions per clock cycle. On-chip caches include two 8 KB direct-mapped data and instruction caches as well as a 96 KB, 3-way set-associative unified second level cache. Support for an external third-level cache with up to 64 MB is included. Virtual address space is 43 bit large, while only 40 bit are implemented physically. Apart from the processor, each node contains up to 2 GB of local memory, a router and other supporting circuitry. Instead of an external third-level cache, stream buffers are used to speed up access to local memory via prefetching on previously detected access patterns.

Access to remote memory, which is non-cacheable, is performed via a large number (512 user + 128 system) of E-registers. A remote load is performed by specifying the target address and an E-register for the result. The result can then be collected from the E-register by a load instruction, which will stall if the result is not yet available. The E-registers significantly increase the number of outstanding loads, since the processor alone can only sustain two outstanding loads. The E-registers therefore provide the necessary support for hiding the latency of remote memory. Unfortunately, the Cray shared memory programming environment ShMem uses synchronous loads, which requires us to implement our own set of communication routines in order to fully utilize the E-registers capabilities.

For a 2048 processor system, the average and maximum network latency is approximately 1500 and 2500 ns, respectively. At a processor speed of 600 MHz, this translates to 900 and 1500 clock cycles, respectively [ST96]. These times exclude memory access and processing at the processing nodes and are therefore quite optimistic. We performed our implementation on the T3E installed at HLRS (Höchstleistungsrechenzentrum Stuttgart) in Germany. This machine has 512 processors running at 450 MHZ and 64 GB of memory (128 MB per processing node).

3.2 Tools

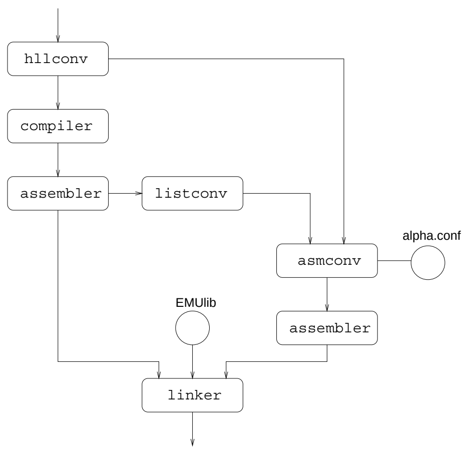

Fig. 2. Design Flow

Several tools were developed to facilitate emulation of multithreading. The design flow and interaction between these tools and the standard programming environment (compilers, assembler, linker) is shown in Fig. 2. The program source code is first modified by the high-level language converter *hllconv*, which searches the source file for user-supplied function names and modifies only those functions. This allows emulation of multithreading to be applied on a function-by-function basis, i.e. only on those functions that benefit from the emulation. The modification process works recursively through all of the given functions. All modified functions are duplicated before the thread-related function calls are substituted with the corresponding calls to the emulation library *EMUlib*. This duplication is necessary since functions may be emulated or executed depending on the caller. The modified source code is forwarded to the compiler, while the list of all modified functions is forwarded to the assembler converter *asmconv*.

The modified source file is compiled using one of the aforementioned compilers, the resulting assembler source is forwarded to the standard assembler *cam*. If there are no functions to be modified, an object file is generated for later use of the standard linker *cld*. Otherwise the assembler listing output is converted to a valid assembler source by *listconv* before it is passed to the assembler converter. The use of the assembler listing instead of the assembler source allows us to ommit support for the advanced *cam* features (e.g. macros) in the assembler converter.

The converted assembler source is processed by the assembler converter, modifiying only those functions given by *hllconv*. If one of those functions is found, the necessary instruction blocks are generated based on the assembler source, the selected grainsize/optimization level as well as the configuration file. This configuration file contains information about instruction styles, modification rules for instructions as well as characteristics of external calls. Since these largely machine-dependent informations are contained within the configuration file, porting of the assembler converter to a different architecture is quite simple.

The assembler converter supports fixed and variable grain-sizes up to several hundred instructions per block. Therefore the number of context switches can be tailored to the application's requirements. In addition, several optimizations that improve instruction scheduling within blocks are available. Calls to external functions are recognized and treated accordingly.

After the whole source file has been processed, the assembler generates an object file from the modified assembler source. After all required object file are generated, the standard linker *cld* combines these object files with the standard libraries as well as the emulation library *EMUlib*. This library contains the thread-related functions for emulation of multithreading, i.e managment, communication and synchronization of threads. The resulting executable can be executed in the usual way.

3.3 Portability

Emulation of Multithreading currently only supports the Alpha architecture. As we already mentioned, this architecture provides a good match for our approach. However, emulation of multithreading is not restricted to the Alpha architecture and can be ported to other architectures. These architectures should meet the following requirements: First of all, it should be possible to load/store the program counter from/to the general-purpose registers. A jalr (jump and link register) type of instruction is sufficient, i.e. is is not required that the program counter is one of the general-purpose registers. For performance reasons, the architecture should support hints to branch prediction logic for this type of instruction, i.e. allow the program counter to be pushed/popped to/from the return-address stack depending on the opcode. The Alpha architecture solves this problem by providing 4 different jalr-type instructions that differ only in the hints to the branch-prediction logic.

Second, the architecture has to provide a minimum number of 6 registers. As we already mentioned above, the main loop uses 3 different registers (framePC, threadPC, returnPC). These registers should be callee-save, otherwise they must be saved/restored upon entry/return to/from external calls, e.g. calls to the operating system. Up to 3 additional registers are required for instruction execution, i.e. to store the operands and results of a single instruction. A larger number of available registers is beneficial for grain-sizes larger than one, since the number of register load/stores from/to the frame can be reduced.

Last but not least, the execution of instructions should depend only on the general-purpose register values, i.e. not on special registers or condition codes. If the architecture has special registers, the load store of these registers from/to the general-purpose registers must be supported. The number of special registers in connection with the number of instructions that depend on these registers affect the size of the frame as well as the average length of the generated subroutines.

In the case of parallel architectures, the hardware must support asynchronous communication between different nodes as well as multiple outstanding communications per node. Since massively parallel computers tend to have large communication latencies, some form of these features is likely to be supported. For performance reasons, the time required to initialize a communication should be small compared to the communication latency.

4 Measurements

The functionality of the assembler converter and the emulation library was tested via several small programs. These programs cover important milestones in the design process, e.g. recursion, external calls and parallelism. After these programs could be successfully modified and executed, we started implementation of a parallel matrix multiplication algorithm.

The program first checks the command-line parameters t (number of threads) and n (matrix size), which must be a multiple of t times p (numer of processors), before shared memory for the three matrices A, B and C is allocated via *shmalloc()*. Matrices A and C are stored in row-order, while matrix B is stored in column-order. The rows and columns are mapped in blocks of t rows resp. columns to the processors in a round-robin manner. Each processor stores random integer values to the corresponding elements of matrices A, B after the random number generator was initialized with the processor's id. After the matrices have been allocated and initialized, a barrier synchronization is performed among all processors before the matrix multiplication begins.

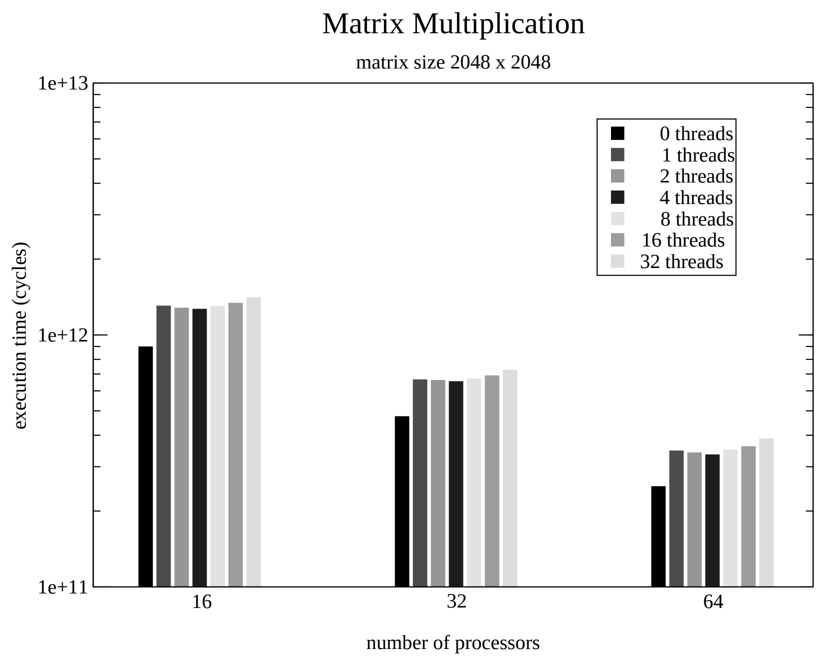

Fig. 3. Experimental Results

The matrix multiplication uses emulation of multithreading, therefore some initialization is required before the multiplication routine can be called. This routine uses the standard matrix multiplication algorithm and operates on all elements mapped to the given thread/processor combination. The elements are read from matrices A, B via *shmem_get()*, while the result is written to matrix C via *shmem_put()*. As we mentioned before, *shmem_get()* works synchronously, i.e. the routine returns only after the desired data has been placed in the target location. Therefore no latency hiding is possible between different invocations of *shmem_get*, i.e. there is no benefit to the emulation of multithreading in this case.

In order to maximize the overhead associated with emulation of multithreading, a fixed grain-size of 1 was used, i.e. context was switched after every instruction. The values reported below therefore provide an upper bound on the overhead associated with emulation of overhead on this application. The execution time of the conventional (unmodified) program was measured as well and provides the baseline for the comparison below. In both cases, the execution time of the matrix multiplication was measured and averaged among all processors. In the case of emulation, this time includes the overhead associated with initialization of threads as well as the matrix multiplication itself.

We performed several experiments with a matrix size of 2048 x 2048 elements and configurations of 16, 32 and 64 processors. For each configuration, the conventional program as well as the multithreaded version using 1, 2, 4, 8, 16 or 32 threads was executed. The results are depicted in Fig. 3. The black bar labeled "0 threads" represents the conventional program and has the smallest execution time. As we explained earlier, this comes as no surprise, since there are no benefits by the emulation of multithreading without latency hiding.

The execution times of the multithreaded programs are quite similar, but increase slightly with the number of threads. Overall, the multithreaded programs run about 1.4 times slower than the conventional version. This is an encouraging result, since the overhead of emulation is smaller than expected. After the emulation library has been extended with asynchronous communication in order to support latency hiding, we are going to repeat these experiments using latency hiding. As the latency to remote memory increases with machine size, we will extend our investigations to larger ($128 \leq 512$ PEs) configurations, since the benfits of emulation increase with latency and number of threads [GK98].

5 Conclusions

Commercial off-the-shelf microprocessors support neither fine- nor coarse-grain multithreading in hardware. As corresponding features (e.g. multiple contexts) have not been announced for any of the mainstream microprocessor families, this situation will not change in the near future. Therefore implementations of multithreading traditionally favored the design of custom microprocessors, but the results were seldom comparable to commercial products. The concept described in this paper utilizes the capabilities of modern microprocessors to enable fine-grained multithreading without hardware support. After development of the basic concept, we exemplified an implementation on the Cray T3E platform. This paper-and-pencil implementation provided the basis for a first analysis of emulating multithreading [GK98]. A grant of computing time from the HLRS allowed us to develop a real implementation.

We successfully implemented the critical tools, i.e. the assembler converter and the emulation library. In both cases development is complicated by the low-level nature of these programs. The assembler converter is quite mature and lacks only instruction scheduling optimizations in the case of larger grainsizes. The converter has been ported to the SimpleScalar architecture [BA97] to facilitate microarchitectural investigations, while the emulation library is currently complemented with asynchronous shared-memory communication and synchronisation primitives. The extended library will allow the porting of whole applications like *fview*, a graph distortion program and *rayo*, a ray tracing program. Afterwards we plan to port some of the SPLASH2 [WOT$^+$95] applications and kernels. A detailed investigation using the aforementioned programs will provide a better view about the capabilities of emulation of multithreading. These upcoming developments will require significantly more resources, since our main focus are large-scale (512 processing elements) configurations.

References

[AKK$^+$93] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[BA97] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, University of Wisconsin-Madison, Computer Sciences Department Technical Report no. 1342, June 1997.

[BH95] Gregory T. Byrd and Mark A. Holliday. Multithreaded processor architectures. *IEEE Spectrum*, 32(8):38–46, August 1995.

[GK98] Andreas Grävinghoff and Jörg Keller. How to Emulate Fine-Grained Multithreading. In *Proceedings of the 2nd IASTED Conference on Parallel and Distributed Computing and Networks*, pages 584–589. ACTA press, 1998.

[KPS94] J. Keller, W.J. Paul, and D Scheerer. Realization of PRAMs: Processor design. In *Proceedings of the 8th International Workshop on Distributed Algorithms (LNCS 857)*, pages 17–27, September 1994.

[Moo96] Simon W. Moore. *Multithreaded Processor Design*. Kluwer Academic Publishers, Norwell, MA, 1996.

[ST96] Steven L. Scott and Gregory M. Thorson. The Cray T3E network: Adaptive routing in a high performance 3D torus. In *Proceedings of HOT Interconnects IV*, August 1996.

[WOT$^+$95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Consideration. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36. ACM, 1995.